MASTER EN SCIENCES INFORMATIQUES

Implémentation et évaluation expérimentale de deux algorithmes de point fixe

Naliba, Marina

Award date:
1995

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

IMPLEMENTATION ET EVALUATION EXPERIMENTALE DE DEUX ALGORITHMES DE POINT FIXE

Promoteur
Baudoin Le Charlier

Mémoire présenté par
Marina Naliba
en vue de l'obtention
du titre de
licencié et maîtrise en iformatique

Année académique 1994-1995

RESUME

Les algorithmes de calcul du point fixe sont destinés à résoudre les problèmes d'analyses statiques, en appliquant la méthodologie de l'interprétation abstraite consistant à remplacer le domaine concret par un domaine abstrait.
Dans le cadre de ce mémoire, on réalise l'étude expérimentale du comportement de deux algorithmes. Le premier représente la classe des algorithmes ascendants, le second appartient à la classe des algorithmes descendants.
Ces algorithmes de calcul du point fixe sont implémentés au départ d'un interpréteur conçu spécialement en vue d'effectuer leur construction.

ABSTRACT

The purpose of the fixpoint computation algorithms is to solve problems in static analysis by applying the methodology of abstract interpretation. It consists in replacing the concrete domain by an abstract domain.
In this work, we pursue the experimental study of the behaviour of two algorithms. The first one describes the class of the bottom-up algorithms and the second one belongs to the class of the top-down algorithms.
The implementation of these fixpoint computation algorithms is based on an interpretor especially designed to construct them.

Qu'il me soit permis d'exprimer toute ma gratitude à Monsieur B. LE CHARLIER, pour avoir accepté, en dépit d'un emploi du temps très chargé, d'être le promoteur de ce mémoire. Son accueil bienveillant, ses judicieux conseils dispensés durant la rédaction de cet ouvrage furent pour moi, un précieux appoint.

Je tiens également à remercier mes amis A. de BAENST, G. LEYDEN et M. PHILIPPE pour leur amitié et leur aide apportée au cours de la révision du manuscrit.

TABLE DES MATIERES

INTRODUCTION

Les algorithmes de calcul de point fixe sont largement appliqués dans plusieurs domaines de programmation, notamment pour l'interprétation abstraite. L'idée de l'interprétation abstraite est de remplacer le domaine de calcul concret par un domaine abstrait possédant les propriétés utiles d'un domaine concret. L'interprétation abstraite est utilisée pour systématiser les analyses statiques appliquées en programmation. Etant introduit pour la programmation impérative classique, l'interprétation abstraite fut ensuite élargie pour la programmation déclarative.

Parmi les algorithmes de calcul de point fixe connus à présent, on remarque deux principales classes des algorithmes, des algorithmes ascendants et des algorithmes descendants. L'objectif de ce mémoire est de comparer expérimentalement deux algorithmes représentatifs de ces classes.

La description de classification des algorithmes de calcul de point fixe, l'aperçu général de chaque classe d'algorithmes et une description plus détaillée avec des exemples concrets des algorithmes construits seront donnés dans le chapitre 2.

Afin de faciliter la compréhension du vocabulaire utilisé dans ce travail, nous donnons les définitions des termes dans le chapitre 1.

L'algorithme classique LINCLOSURE développé pour l'hyper-graphe donna naissance à l'algorithme ascendant [13] destiné à résoudre des problèmes d'analyse statique de programmes. Pour avoir une idée d'application de cet algorithme, on peut confirmer que le problème de gestion de dépendance fonctionnelle dans la base de données relationnelle peut être résolue grâce à l'algorithme ascendant de calcul de point fixe. L'algorithme ascendant s'applique à un système d'inéquations monotones.

L'algorithme général de calcul de point fixe pour une transformation possédant la propriété d'être monotone a été introduit par Le Charlier et Van Hentenryck [12] c'est un exemple d'une méthode descendante. L'accent a surtout été porté sur la construction d'un algorithme efficace permettant de réduire le nombre de calcul autant que possible. L'idée d'utiliser le graphe de dépendance a permis d'éviter les calculs redondants par l'algorithme général de calcul de point fixe.

Nous présentons la mise en oeuvre de l'algorithme ascendant et de l'algorithme descendant conçus pour une transformation décrite par un système d'inéquations dans le chapitre 5.

Pour pouvoir mettre en oeuvre des algorithmes de calcul de point fixe pour un système d'inéquations, on aura besoin d'écrire ce système d'inéquations sous une forme ou l'autre. Le langage SYSTINEQ est conçu pour présenter un système d'inéquations à étudier avec une syntaxe lisible par ordinateur. La syntaxe du langage SYSTINEQ sera abordée au chapitre 3.

L'interpréteur d'un programme SYSTINEQ est le moyen de donner à partir d'un système d'inéquations décrit avec la syntaxe du langage SYSTINEQ une représentation interne sous forme

de pile d'inéquations reconnaissable par les algorithmes de calcul de point fixe. Toutes les étapes de passage intermédiaires entre un programme SYSTINEQ et la représentation finale sous forme de pile sont décrites dans le chapitre 4.

Pour plus de détails sur la mise en oeuvre de l'interpréteur et sur la programmation des algorithmes, le lecteur plus intéressé pourra consulter les annexes.

Après avoir présenté deux algorithmes de calcul de point fixe et de l'interpéteur à la base duquel les algorithmes ont été conçu, nous examinerons, dans le chapitre 6, le comportement des algorithmes sur des exemples spécialement construits pour démontrer des points forts et des 'faiblesses' des algorithmes présentés.

Dans le chapitre 7 nous donnons le mode d'emploi du programme contenant deux parties principales: celle de l'interpréteur et celle des algorithmes ascendant et descendant. Des exemples des programmes SYSTINEQ étudiés par le chapitre 6 dans le cadre d'étude expérimentale sont également fournies.

Chapitre 1. DEFINITIONS

Le but de ce chapitre est d'essayer de trouver une suite de termes qui conviennent au vocabulaire que nous utiliserons.

1.1. Ensemble

cardinal

- Le cardinal d'un ensemble fini est le nombre des éléments.

Exemple :

|{1,2,3}|=3.

puissance

- La puissance d'un ensemble est l'ensemble de tous les sous-ensembles de cet ensemble .

A titre de notre exemple :
P({1,2,3]})={{},{1},{2},{3},{1,2},{1,3},{1,2,3}}.

mapping

- L'ensemble de tous les mapping de **A** en **A**, **MAP(A)** est l'ensemble de toutes les fonctions partielles ou totales de **A** en **A**.

Exemple:

|**MAP**({1,2,3})|=64.

1.2. Ordre partiel

La théorie de treillis est construite autour d'une relation binaire R qui peut être lue comme "est compris dans", " fait partie de ", "plus petit ou égale à". Cette relation binaire doit répondre à certaines propriétés pour déterminer le concept de l'ensemble de l'ordre partiel (poset).

ordre partiel(po)

•L'ordre partiel de l'ensemble **A** est une relation **R** qui est définie pour**A.** Cette relation est

transitive : si x**R**y et y**R**z alors x**R**z;

antisymétrique : si x**R**y et y**R**x alors x=y;

réflexive : pour tout x de **A** x**R**x.

Exemple:

$\subseteq$ pour **P**({1,2,3}) est une relation d'un ordre partiel, elle est

transitive : si x$\subseteq$y et y$\subseteq$z alors x$\subseteq$z;

antisymétrique : si x$\subseteq$y et y$\subseteq$x alors x=y;

réflexive : $\forall$ t de **P**({1,2,3}), t$\subseteq$t.

On appelle l'ordre n(**A**) de l'ensemble **A** avec la relation de l'ordre partiel **R** définie[1] le cardinal de cet ensemble.

Si le nombre des éléments de <**A**, **R**> est fini, alors on dit que le <**A**, **R**> est fini.

ensemble ordonné

• Un ensemble ordonné <**A**, **R**> est un ensemble muni d'une relation d'ordre **R**.

majorant

• Un majorant de **X** est un élément de **A** tel que, pour tous x de **X**, on ait x**R**a.

minorant

• Un minorant de **X** est un élément de **A** tel que, pour tous x de **X**, on ait a**R**x.

---

[1]On utilise la notation <**A**, **R**> pour designer un ensemble ordonné **A**, l'ordre partiel est défini par **R**

Il ne faut pas confondre des concepts de minorant et de majorant avec des concepts des éléments minimaux et maximaux.

maximum

- Le maximum de **X** est un majorant de **X** qui appartient à **X**.

On montre facilement que , s'il existe, il est unique.

minimum

- Le minimum de **X** est un majorant de **X** qui appartient à **X**.

On montre facilement que , s'il existe, il est unique.
On voit que le minorant peut être minimal , et le majorant peut être maximal, mais le contraire n'est pas vrai.

la borne supérieure(Lub)

- La borne supérieure de **A** est le minimum, s'il existe, des majorants de **A**. On la note **Lub(A)**.

Exemple:

{1,2,3} est un élément maximal de {{1},{2}}, et {1,2} est la borne supérieure .

la borne inférieure(Glb)

- La borne inférieure de **A** est le maximum, s'il existe, des minorants de **A**. On la note **Glb(A)**.

ensemble complet(cposet)

- Un ensemble ordonné <**A**, **R**> est dit complet (ou plus précisement sup-complet), si toute partie de **A** admet une borne supérieure.

Si un ensemble ordonné est complet , la partie vide a aussi une borne supériure , on le note $\perp$ (bottom). Puisque l'ensemble des majorants de la partie vide est **A** tous entier, $\perp$ est le minimum de **A**. Le fait qu'un ensemble soit complet implique donc qu'il contienne un élément minimum.

1.3. Treillis

chaîne croissante

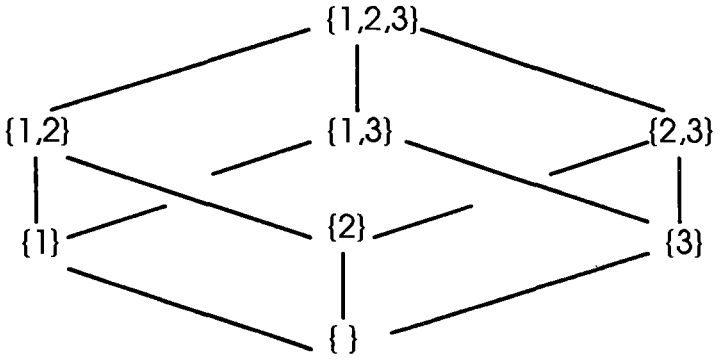
- La chaîne croissante est une séquence des éléments de **X** tel que x_1 **R** x_2 **R** x_3 **R** ...

On peut toujours définir la longueur d'une chaîne, si elle est finie. De façon générale, la longueur de **<A, R>**, noter comme l(**A**), peut être définie comme la borne supérieure des longueurs des chaînes croissantes dans **<A, R>**.
Si l(**A**) est fini, alors on dit que l'ensemble **A** avec la relation de l'ordre partiel défini a une longueur finie.
Dans l'ensemble ordonné **A** l'hauteur ou dimension h[x] de chaque élément $x \in A$ est la borne supérieure (**Lub**) des longueurs de chaînes $x_0<x_1<...<x_l=x$ entre x_0 et x.

treillis(lattice)

- Un ensemble ordonné **A** est un treillis (lattice) si toute paire d'éléments admet une borne supérieure(**Lub**) et une borne inférieure(**Glb**).
- Un treillis **T** est complet si tout ensemble **X** a la borne supérieure et la borne inférieure dans **T**.

Exemple:



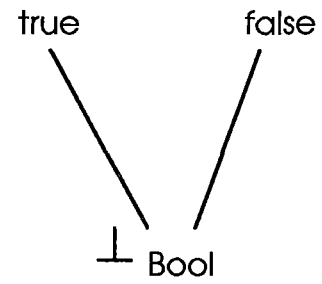
<P({1,2,3}) ,⊆> est un treillis complet.

On utilise la notation ⊥(bottom) pour dénoter la borne inférieure de treillis (**Glb(T)**).
La construction d'une structure de treillis consiste à ajouter la borne inférieure ou un élément indéfini ⊥ (bottom) dans la relation de l'ordre partiel(po).

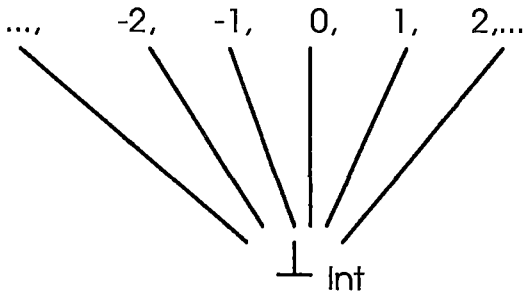
Exemple:

On peut définir un treillis complet avec l'ordre partiel **R** pour les domaines élémentaires suivants:

le domaine Booléene:



le domaine des entiers:



On voit que la construction d'un treillis consiste à ajouter un élément indéfini ⊥. Une borne inférieure ⊥ pour chaque domaine peut être un élément entièrement indéfini et non spécifié.

1.4. Fonctions

treillis des fonctions

On a défini ci-dessus la notation des treillis des éléments d'un ensemble ordonné avec un ordre partiel sous cet ensemble.

Des éléments d'un ensemble peuvent être des fonctions.

- Si **T= <A, R>** est le treillis complet, on définit **R'** pour **MAP(A)** et on dit que **fR'g** si et seulement si f(x)**R**g(x) ∀x:**A**.

  Alors **R'** est un ordre partiel pour **A→A**:

  transitive : si **fR'g** et **gR'h** alors **fR'g**;

  antisymétrique : **fR'g** et **gR'f** alors f=g;

  réflexive : f(x)**R**f(x) ∀ x:A alors **fR'f**.

L'ensemble de tous les mappings de **A** dans **A, MAP(A), R** forme un treillis complet.

fonction continue

- Soit **T** est le treillis complet et f:**T→ T** est "mapping" alors on dit que f est continue si f(**Lub**(x))= **Lub**(f(x)) pour tous les sous-ensembles **X** de **T**.

fonction monotone

- Une fonction monotone f de treillis **T=<A, R>** est celle pour laquelle le fait que **xRy** implique le fait que f(x)**R**f(y).

fonctions partielles

Pour représenter les fonctions monotones partielles, on utilise la notation **A⇒A'**.

On dit que f(x) =undef si une fonction partielle f est indéfinie pour valeur x∈ **A**. Le domaine d'une fonction partielle, dom(f), est un ensemble de x∈ A tel que f(x)≠undef.

On peut introduire une relation de l'ordre partiel ⊆ sur l'ensemble des fonctions monotones partielles:

f≤g si et seulement si dom(f)⊆dom(g) &

f(α)≤g(α) pour tout α∈ dom(f).

1.5. Point fixe

Nous sommes arrivées vers une définition cruciale de notre travail, celle du point fixe.

point fixe

- Soit **T=<A, R>** est le treillis complet et f:**A**→**A** est "mapping", alors μ(f) est le point fixe d'une fonction f si μ(f)=f(μ(f)).

le plus petit point fixe

- Le point fixe de fonction f:**A**→**A** défini sur le treillis complet **T=<A, R>** est le plus petit point fixe si pour tous les points fixes a de f on a μ(f)≤a.

Le plus petit point fixe est égal à la limite d'une suite croissante d'approximations:

f^0**R...R**f^n**R...**

On a introduit assez de concepts pour démontrer lé théorème du point fixe.

Théorème du point fixe:

Enoncé:

Soit (**A**, **R**)est un treillis, f est une fonction continue de **A** vers **A**.
Alors f admet un plus petit point fixe μ(f) unique qui vérifie

$\mu(f)=$**Lub**$(f_n(\perp))$.

Démonstration:

La suite $(f^n(\perp))_{n\geq 0}$ est une chaîne avec $\perp=f^0(\perp)$.

Soit x=**Lub**$(f^n(\perp))$. Comme f est continue,

$f(x)=$**Lub**$(f^{n+1}(\perp))$

et puisque $\perp=f^0(\perp)$ est le minimum de **A**,

Lub$(f^{n+1}(\perp))=$**Lub**$(f^n(\perp))=x$

qui est donc un point fixe de f. Si y est un autre point fixe de f, on démontre d'abord par induction que $\forall n\in$ **N**, $f^n(\perp)\leq y$; puisque $\perp$ est minimum de **A**, $\perp=f^0(\perp)\leq y$; si $f^n(\perp)\leq y$ alors $f^{n+1}(\perp)\leq f(y)=y$. D'où il découle que x=**Lub**$(f^n(\perp))\leq y$.

1.6. Transformations

Pour ce travail, il est très utile de parler de définition de point fixe d'une transformation quelconque τ.

On utilise la notation $A \rightarrow A'$ pour représenter l'ensemble des fonctions monotones de A dans A', où A et A' sont des cpos.

De façon analogue, $(A \rightarrow A') \rightarrow (A \rightarrow A')$ est un ensemble de fonctions monotones qu'on appelle "transformations". On utilise la lettre τ pour les dénoter.

le plus petit point fixe d'une transformation

- On détermine le plus petit point fixe de la transformation τ comme $\mu(\tau)$.
  Autrement, on peut dire que le plus petit point fixe de τ est égal à la limite d'une chaîne croissante d'approximations:

  $$f^0R \ldots Rf^kR \ldots .$$

Chapitre 2. ALGORITHMES DE CALCUL DE POINTS FIXES

2.1. Interprétation abstraite

Avant d'aborder le problème du calcul des points fixes et des algorithmes développés pour résoudre ce problème, on donne la description du domaine de travail. On parle de l'analyse statique et de l'interprétation abstraite[8].

> " Le terme d'analyse statique de programmes recouvre l'ensemble de tous les traitements que l'on peut appliquer à un programme en dehors de son exécution proprement dite, par exemple: au cours de la compilation."

Entre autres, cela peut être l'information pour faire la transformation, afin d'optimiser un programme. La technique de l'analyse statique est utilisée de longue date dans les compilateurs sans faire appel à la théorie classique. Prouver qu'une méthode d'analyse statique est correcte dans tous les cas est un problème assez difficile, parce que les langues de programmation sont complexes à analyser.

Le terme d'interprétation abstraite était introduit par P. et R. Cousot [4]. L'interprétation abstraite est une base mathématique qui exprime les méthodes existantes et prouve de manière systématique leur correction.

L'interprétation abstraite se base sur le fait que l'on remplace le domaine du calcul normal par un domaine "non standard" ou "abstrait" sous des hypothèses respectées:

- ◊ les éléments du domaine abstrait représentent des propriétés utiles du domaine standard;
- ◊ les calculs sur le domaine abstrait peuvent être réalisés de manière suffisamment efficace et convergent en un temps fini.

Par exemple, dans le cas de multiplication algébrique, on peut remplacer l'ensemble des nombres qui représentent notre domaine concret par l'ensemble "abstrait" des signes: {+,-}.

On introduit les règles suivantes pour la multiplication des nombres:

+*+=+

+*-=-

-*+=-

-*-=+

Cet exemple de règles permet de calculer le signe du produit de multiplication sans réellement faire le calcul.

> "Des recherches actuelles visent à définir des méthodes systématiques de validation des domaines abstraits. Une autre voie de recherche prometteuse est l'élaboration de systèmes génériques basés sur l'interprétation abstraite.
>
> Ces systèmes seraient paramètrés selon trois critères:

◊ la sémantique du point fixe du langage à analyser,

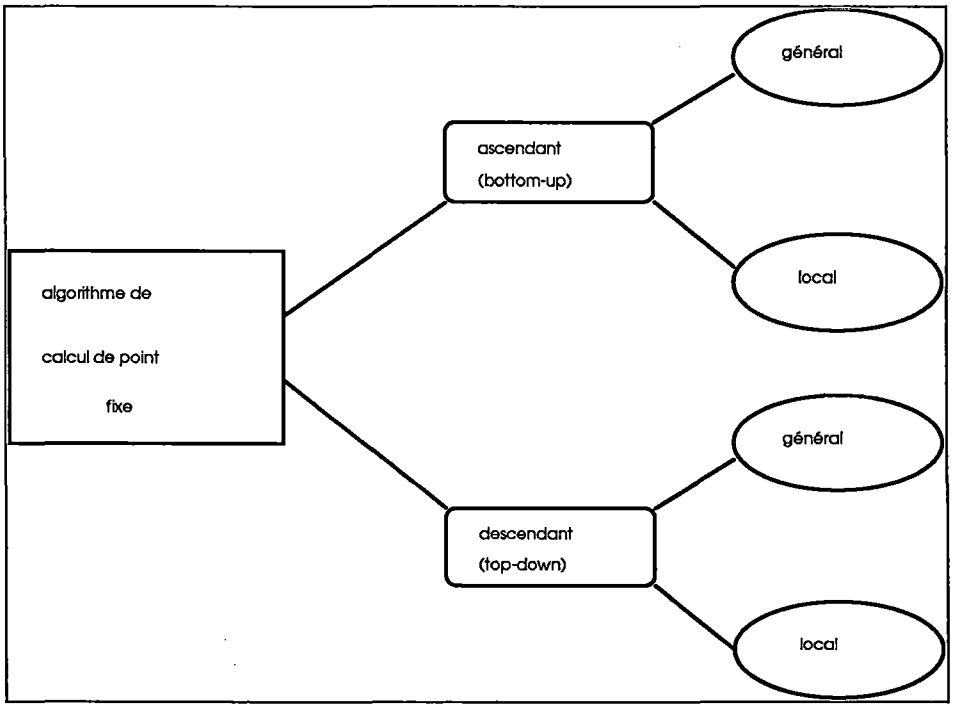
◊ le domaine abstrait,

◊ l'algorithme de calcul du plus petit point fixe."[8]

Une sémantique abstraite n'est pas nécessairement seule et unique. En effet, à partir d'une sémantique concrète, on peut dériver plusieurs sémantiques abstraites.

2.2. Classification des algorithmes de point fixe

La variété des algorithmes existants de calcul de point fixe utilisés en interprétation abstraite peut être représentée par le schéma suivant:



Ils peuvent être divisés en algorithmes ascendants et descendants. Chaque méthode peut être développée à son tour pour l'algorithme général et local.

On va procéder de la façon suivante: on donne la description d'une méthode ascendante pour deux algorithmes concrets développés pour un système d'équations. Le premier d'entre eux utilise le principe d'approximation chaotique du point fixe. Ensuite, la description intuitive d'une méthode descendante sera présentée. La méthode descendante est illustrée par l'exemple de l'algorithme général de calcul de point fixe de B. Le Charlier. Cet algorithme est valable pour toute transformation quelconque; on exige seulement la propriété d'être monotone de transformation. L'algorithme local de méthode descendante, inspiré de l'algorithme général et construit pour le système d'inéquations, est décrit à la fin du chapitre.

2.3. Système d'équations monotones

Plusieurs des algorithmes de calcul de point fixe proposés sont développés pour la transformation ayant la forme de l'ensemble d'équations.

On est dans l'environnent de l'ensemble ordonné **A** qui forme la structure de cpo d'hauteur finie et possédant un élément minimal $\perp$.

Le système d'équation **E** défini sur cpo<**A**, **R**> a la forme[2] :

$$\begin{cases} x_1 = f_1 \\ \quad . \\ \quad . \\ \quad . \\ x_n = f_n \end{cases}$$

où

- la partie gauche de l'équation est une variable $x_i \in \mathbf{X}$,
- $\mathbf{X}=\{x_1, ..., x_n\}$,
- la partie droite de l'équation est une fonction $f_i:(\mathbf{X}\rightarrow\mathbf{A})\rightarrow\mathbf{A}$,
- les variables x_i sont supposées différentes $x_i \neq x_j \ \forall\ 0< i,j \leq N$,
- les fonctions f_i sont monotones.

Un élément $\theta \in [\mathbf{X}\rightarrow\mathbf{A}]$ est une solution de **E** ssi

$\theta(x)=f_x(\theta)$ pour tous $x \in \mathbf{X}$,

où f_x est la fonction f de partie droite d'une équation ayant x comme partie gauche.

Les fonctions de partie droite sont monotones; alors, on peut conclure d'après la théorie de point fixe que le système d'équations **E** a la plus petite solution, et qu'il est unique. On le note $\mu(\mathbf{E})$.

---

[2] Dans plusieurs algorithmes qui vont être disposés dans ce travail , on utilise par défaut la transformation sous forme d'un système d' équations présentées dans cette section. Si ce n'est pas le cas chaque fois, on précise la forme de transformation τ.

2.4. Présentation générale

2.4.1. La méthode ascendante (bottom-up)

L'algorithme ascendant traditionnel de calcul de point fixe d'une transformation monotone τ peut être representé de façon intuitive par le schéma suivant: les variables $x_1, \ldots, x_n$ sont initialisées par la valeur de $\perp$; au fur et à mesure du calcul, les valeurs des variables sont recalculées jusqu'au moment où les variables deviennent stables. Le fait que les variables sont stabilisées signifie qu'on a trouvé le point fixe.

La méthode ascendante est organisée de manière à obtenir d'abord toutes les valeurs calculables directement; ensuite, celles qui ne dépendent que des premières déjà calculées et ainsi de suite...

2.4.2. La methode descendante (top-down)

Le point de départ est la définition récursive du point fixe; on cherche à calculer récursivement la valeur $\mu(\tau)$.

Pour organiser le calcul de point fixe par l'algorithme de façon efficace, on a besoin d'une table des valeurs des variables x_i comprises dans la transformation τ.

Cette table contient les valeurs des variables pour lesquelles un appel récursif est déjà lancé; peu importe si cet appel est terminé ou pas. En plus, la table contient une approximation inférieure du résultat $f(x_i)$. On utilise l'approximation trouvée dans la table au cours d'un même appel récursif. A la fin de chaque appel, on améliore le contenu de la table avec la valeur trouvée.

On continue l'itération du même calcul jusqu'à ce qu'aucune amélioration du résultat ne soit constatée.

La technique de calcul descendant de point fixe pour une variable locale porte encore le nom de technique d'induction. Elle est décrite en détails dans [3,6,7].

De façon intuitive, on commence par faire une supposition de valeur de $\mu(\tau)(x)$. Cela signifie que l'on prend une valeur vR $\mu(\tau)(x)$ et que l'on convient que $\mu(\tau)(x)$ est égale à v. Ensuite, on vérifie si la supposition préalable est réellement correcte. Dans le but de contrôler si la supposition est valide ou pas, on évalue $f_x(\mu(\tau))$ en tenant compte que $f_x(\mu(\tau))$ tend vers $\mu(\tau)(x)$, ce qui signifie que la supposition est correcte. Au cours de l'exécution de $f_x(\mu(\tau))$, il peut arriver que l'on ait besoin de quelques composants auxiliaires. Il n'y a aucun problème pour estimer $\mu(\tau)(x)$: en correspondance avec notre supposition, on prend directement la valeur v. Si l'on a besoin de $\mu(\tau)(x')$ pour une certaine $x' \neq x$, alors on lance le calcul de $\mu(\tau)(x')$ en restant toujours dans la même condition que $v = \mu(\tau)(x)$. On peut utiliser l'application récursive de schéma ci-dessus. Au son tour, on suppose que $v' = \mu(\tau)(x')$ et on essaie de prouver que l'évaluation de $f_{x'}(\mu(\tau))$ nous donne v'. La différence de cette étape par rapport à l'étape précédente se trouve dans le fait qu'ici, on est dans les conditions de deux suppositions, la première: $v = \mu(\tau)(x)$ et la seconde: $v' = \mu(\tau)(x')$.

Pour calculer la valeur de $f_{x'}(\mu(\tau))$, on peut avoir besoin de l'autre composant $\mu(\tau)(x'')$ $(x \neq x' \neq x'')$. Nous appliquons à nouveau notre schéma récursif et ainsi de suite...

2.5. L'approximation chaotique

On va exposer la méthode ascendante dans le cadre de calcul de point fixe par l'approximation chaotique [14]. La méthode d'itération chaotique était introduite par P.Cousot dans [5] pour une large classe de méthodes. Cette méthode sert à trouver la plus petite solution pour l'ensemble d'inéquations. De façon générale, l'efficacité de cette méthode dépend forcément du problème à résoudre.

2.5.1. L'approche générale

L'algorithme calcule le point fixe pour un systéme des équations E. Il est général: le point fixe est calculé pour toutes les variables comprises dans le systéme. Le principe de l'algorithme est démontré par le schéma suivant:

```
val:=⊥ {initialisation}
do not (∀x∈ A:val(x)=fx(val))→
    x:=choose(X)
    val(x):=EvalRhs(x,val)
od
    val=μ(E)
```

La structure des données val:**X→A** contient la valeur courante des variables.
L'algorithme choisit la variable, évalue la partie droite correspondante et, au fur et à mesure des calculs appropriés, modifie la valeur d'une variable associée.
La fonction d'évaluation *EvalRhs* est assumée pour évaluer les parties droites. Le processus de modification se termine quand toutes les variables deviennent stables.
Pour rendre un algorithme de calculs plus efficace, l'opération *choose* doit être développée de telle sorte que le nombre d'itérations requis pour atteindre la solution minimale soit le plus bas possible.
L'idée générale de stratégie de sélection d'une variable est la suivante:
Supposons que la variable x soit choisie pour évaluer. Evidemment, si l'on a reçu au cours des exécutions que l'évaluation de f_X(val) donne val(x), cela signifie que le choix de x est un mauvais choix. On a dépensé du temps pour calculer f_X(val) sans faire aucun progrès. Pour minimiser le temps de calcul, il faut éviter le choix des variables qui sont potentiellement stables. On utilise l'ensemble supplémentaire qui tient la trace des variables qui "promettent" d'être instables ws∈ A.

Les variables qui ne sont pas comprises dans l'ensemble ws sont des variables stables.
De façon formelle

(*)$\forall x \in X \mid$ ws: $f_X(val)=val(x)$.

Les variables appartenant à ws et seulement à elles doivent être choisies dans un but de réévaluation. On peut déduire la propriété de ws:si ws=∅ alors val est stable. La proposition (*) est en effet l'invariante; nous l'appelons I_{WS}. On a intérêt à avoir la taille |ws| la plus petite possible. L'invariante Iws doit être raffinée chaque fois que les valeurs des éléments du tableau val changent suite aux résultats de la réévaluation de la partie droite du système d'équations.
La fonction supplémentaire *Restore(x,ws)* est ajoutée pour tenir compte des dépendances entre des variables et les parties droites d'un système d'inéquations.
En ajoutant la fonction *Restore*, on obtient le schéma modifié qui suit:

```
{initialisation}
val:=⊥
ws:=X
{fin de l'initialisation}
while ws≠∅ →
        x:=choose(X)
        ws:=ws\{x}
        vold:=val(x)
        val(x):=EvalRhs(x,val)
        if vold≠val(x)→Restore(x,ws) fi
od
val=μ(E)
```

Les auteurs proposent deux solutions de construction de fonction *Restore(x,ws)*.
La première solution utilise l'information de dépendance syntaxique. De façon intuitive pour chaque variable, on cherche un ensemble des variables duquel celle-ci dépend .
Cette information peut être trouvée simplement en parcourant la structure de données, puis on enregistre les variables rencontrées. Cette dépendance est statique dans le sens qu'elle est déterminée une fois pour toutes et n'est pas changée au cours des évaluations.
La seconde approche a pour but d'utiliser la dépendance sémantique. De façon informelle, on enregistre l'ensemble des variables duquel notre variable x dépend au cours des exécutions. Cette information est dynamique parce qu'elle change avec le temps. Le problème d'une dépendance dynamique est discuté par les auteurs dans [5,9,10].
L'utilisation des dépendances dynamiques devient intéressante si la partie droite des équations est 'sparse'. On dit que f est 'sparse' si pour l'environnement θ, seulement le petit fragment de θ est

nécessaire pour évaluer f(θ). Souvent, un système d'équations booléens a un degré élevé de 'sparsness'(true∪_=true, false∩_=false).
Les notions de dépendance syntaxique (statique) , de dépendance sémantique (dynamique), de 'sparsness' sont très importantes pour comprendre le reste de la présentation des algorithmes.

2.5.2. L'approximation locale

L'approximation des points fixes peut être globale et locale . Elle est globale dans le sens de calculer toutes les solutions minimales. En réalité,on a plus souvent besoin d'une variable particulière; disons x°. Dans ce cas, l'algorithme local cherche la composante μ(**E**)(x°) d'intérêt. Pour trouver cette composante, on n'a pas intérêt à calculer tous les points. L'algorithme local a pour but de développer seulement un fragment nécessaire d'un système d'équations **E**.
La différence essentielle entre l'algorithme global et l'algorithme local se trouve dans le mécanisme de détection de stabilité. A la place de stabiliser à priori toutes les variables comme l'algorithme général le fait, le schéma local essaie de construire seulement le point fixe 'partiel', ce qui signifie qu'elle fait la construction d'un sous-ensemble **S** de variables (l'espace de recherche) qui est stable et complet dans ce sens que les valeurs de variables de **S** ne dépendent que des valeurs des variables intérieures de **S**.

2.6. Algorithme Bottom-Up

L'algorithme Bottom-Up [13] est orienté pour résoudre une certaine classe des problèmes de points fixes définis sur le treillis. Il s'inspire d'un algorithme LINCLOSURE[1]. Il est de nature ascendante; étant général, il calcule le point fixe pour toutes les variables de transformation. L'algorithme est programmé en Pascal, le code-source de l'algorithme Bottom-Up figure dans l'annexe 2.

2.6.1. Environnement

◊ On travaille sur le treillis complet **T**.

◊ Etant donné un ensemble de fonctions croissantes monotones
f_j: **X**→ **T**.

◊ Le problème des inéquations monotones est une ensemble d'inéquations; donc, chacune a une des trois formes suivantes:

$x_k \geq f_k$

$x_k \leq f_k$

$x_k = f_k$

Le problème du calcul du plus petit point fixe d'un système monotone des inéquations peut être résolu en réécrivant les inéquations données avec les règles de réécriture suivantes:

$x_k \geq f_k \Rightarrow x_k \geq f_k$

(**) $x_k \leq f_k \Rightarrow$ true

$x_k = f_k \Rightarrow x_k \geq f_k$

Résoudre le problème de point fixe de systèmes monotones des inéquations signifie trouver un modèle minimal de l'ensemble d'inéquations, où

◊ modèle pour l'ensemble d'inéquations est un ensemble des éléments de treillis tel qu'en remplaçant des variables par ces éléments, on obtient des inéquations justes;

◊ étant donné deux modelés M_1 et M_2 pour l'ensemble d'inéquations, on dit que M_1 est sous-modéle de M_2 si

$(\forall\ 1 \leq i \leq n)\ M_1(xi) \leq M_2(xi)$ &

$(\exists 1 \leq j \leq n)\ M_1(xj) < M_2(xj)$

◊ Le modèle est minimale, si et seulement si elle ne possède que son propre sous-modèle.

D'après la théorie de treillis [2], le point fixe de fonction monotone sur le treillis complet forme un treillis complet. En particulier, il y a un point fixe unique et ce point fixe est un modèle minimal pour l'ensemble d'inéquations.

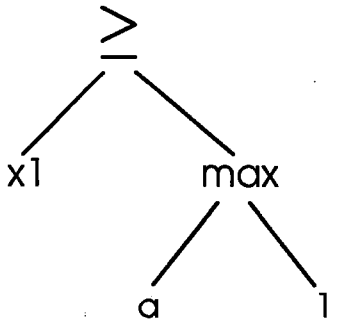
2.6.2. Algorithme

La taille du problème est définie par la somme des tailles des arbres des inéquations.

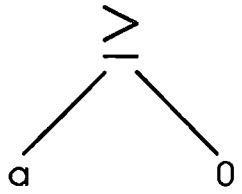
Par exemple, pour un système d'inéquations

1. $x_1 \geq \max(a,1)$ {taille 4}
2. $x_2 \geq *(x_1, b)$ {taille 4}
3. $p \geq +(x_2, c)$ {taille 4}
4. $a \geq 0$ {taille 2}
5. $b \geq 1$ {taille 2}
6. $x_3 \geq *(a, c)$ {taille 4}
7. $c \geq +(b, x_3)$ {taille 4}

Pour la première inéquation, l'arbre fonctionnel a la forme :



Pour la quatrième inéquation, l'arbre fonctionnel représente :



Dans notre exemple la taille du problème égale à 24.

L'algorithme utilise la structure des données suivantes:

N : nombre des variables

E : nombre des inéquations

pour i:=1 to N

value[i] : la valeur courante de variable x_i

chain[i] : la liste de toutes les parties droites où x_i apparait

link[i] : pointe vers un élément suivant dans la pile, égale à NIL si c'est le dernier élément, OUT si x_i n'est pas dans la pile

pour j:=1 to E

lhs[j] : indexe de la variable

f[j] : x[lhs[j]] ≥ f[j] est inéquation de numéro j

Les variables auxiliaires :

s : pointe vers le sommet de la pile

i : l'indexe d'une variable à évaluer

c : pointe vers le reste de la chain[i]

j : indexe d'inéquation à évaluer

t : la valeur de treillis

L'algorithme est divisé dans deux phases:

◊ la phase d'initialisation

◊ la phase de clôture.

La phase d'initialisation calcule la valeur initiale de chaque variable, et la met sur la pile si cette valeur est différente de 0.

La fonction *eval(expr)* évalue l'expression avec le temps qui est proportionnel à la taille d'une expression.

```
for i from 1 to N do
  value[i], link[i] := 0, OUT
od
s:=NIL
for j from 1 to E do
  i:=lhs[j]
  t:=value[i]
  eval(expr[j])
  if t> value[i] then
    value[i]:=t
    link[i]:=s
    s:=i
  fi
od
```

Pour notre exemple, la phase d'initialisation donne

x1:=1

x2:=0

p:=0

b:=1

x3:=0

c:=1

La pile devient

s=[c, b, x1].

Le prix d'une phase d'initialisation est proportionnel à la taille du problème.

Pour que dans une phase d'initialisation, toutes les variables aient des valeurs appropriées, la pile contient toutes les variables avec des valeurs différentes de zéro.

La phase de fermeture produit la propagation des variables changées jusqu'au moment où il n'y a plus de changements de valeurs des variables.

En appliquant la phase de terminaison à notre exemple, on obtient:

→ dépiler c de la pile

évaluer $p \geq +(x_2, c)$

mettre p=1 et empiler p dans la pile

évaluer $x_3 \geq *(a, c)$

pas de changement

→ dépiler p de la pile

p n'apparaît dans aucune partie droite

→ dépiler b de la pile

évaluer $x_2 \geq *(x_1,b)$

mettre $x_2=1$ et empiler x_2 dans la pile

évaluer $c_3 \geq +(b, x_3)$

pas de changement

→ dépiler x_2 de la pile

évaluer $p \geq +(x_2,c)$

mettre p=many et empiler p dans la pile

→ dépiler p de la pile

p n'apparaît dans aucune partie droite

→ dépiler x1 de la pile

évaluer $x_2 \geq +(x_1, b)$

pas de changement.

Le résultat final est

p=many

a=0

b=1

c=1

L'optimisation possible de cet algorithme consiste à éviter d'empiler dans la pile, les variables qui n'apparaissent dans la partie droite d'aucune inéquation.

2.6.3. Complexité

Chaque itération d'une boucle extérieure correspond à un changement de valeur de xi. Consécutivement, quand on parle du coût d'un problème entier, il faut prendre en compte le nombre d'itérations de la boucle extérieure.

Le corps de la boucle intérieure prend le temps constant. La question suivante est le temps d'itérations de la boucle intérieure. On traverse chaque chaîne tout entière chaque fois que la variable correspondante est empilée.

Alors, si on met dmax pour désigner la profondeur maximale de treillis, chaque variable peut être empilée dmax fois. On peut estimer la grandeur de nombre d'itérations comme

$$d_{max} \times \sum_{i=1,\ldots,N} length(chain[i])$$

Mais dmax× $\sum_{i=1,\ldots,N} length(chain[i])$ est le nombre des variables qui apparaissent dans les parties droites, il est borné par la taille du problème.

La complexité totale de l'algorithme est

$O(d+E+N+d_{max}$*taille de problème)

$d \leq d_{max} \times N$ ⇒

E+N< taille de problème

alors, le coût du problème est $O(d_{max} \times$ taille de problème).

2.6.4. Applications

L'algorithme présenté ci-dessus couvre une classe des problèmes de programmation logique et de théorie des graphes.

Voici quelques applications concrètes de cet algorithme:

- ◊ pour manipuler les dépendances fonctionnelles dans la base des données relationnelles;
- ◊ pour la révision dans un système d'expert;
- ◊ dans la programmation logique avec les incertitudes ;
- ◊ pour construire une fermeture transitive;
- ◊ pour analyser des programmes logiques.

2.7. Algorithme général de B. Le Charlier

2.7.1. Problème

L'algorithme général [12] est orienté pour assumer des transformations qui peuvent apparaître comme le résultat des faits de remplacement d'un domaine concret par un domaine abstrait.

L'algorithme calcule le minimal point fixe de sémantique abstraite.

Le domaine de calcul de l'algorithme est cpo. Cpos de travail sont appelées par **A** et **A'**. La notation **A→ A'** est utilisée pour représenter un ensemble des fonctions monotones de **A** en **A'**. De façon analogue (**A→ A'**)→ (**A→ A'**) est l'ensemble des transformations monotones. On utilise la lettre τ pour dénoter les transformations. La définition des fonctions partielles, fonctions partielles monotones, de plus petit point fixe sont données dans la chapitre 1 "Définitions".

Etant donné

- ◊ la transformation τ, définie par procédure tau décrite dans le langage de programmation,
- ◊ le paramètre fonctionnel f,

◊ valeur α,

l'algorithme de calcul de points fixes renvoie la valeur $\alpha f\tau$.

La procédure tau doit posséder la propriété d'être monotone.

L'algorithme est descendant (top-down), il est de nature quasi-directe, ce qui signifie qu'il est focalisé autour de sous-ensemble de points fixes minimaux, nécessaires pour calculer la valeur de α. L'algorithme utilise les dépendances dynamiques entres des valeurs de paramètre fonctionnelle pour éviter les calculs redondants pendant le calcul de point fixe. L'idée générale est de reconstituer seulement des appels de paramètres fonctionnels qui dépendent de façon transitive des autres appels pour lesquels on a constaté une modification des valeurs.

L'algorithme général de calcul de point fixe peut être appliqué aux domaines infinis sous certaines modifications.

2.7.2. Avant présentation

Le but de l'algorithme est d'éviter le calcul d'une table entière de fonction $\mu(\tau)$ pour une raison pratique, parce que d'une part A peut être infinie; et, d'autre part pour calculer $\mu(\tau)\alpha$, il suffit d'avoir un petit sous-ensemble de $\mu(\tau)$. Le résultat d'un algorithme est la table d'une fonction partielle f telle que $f(\alpha)=\mu(\tau)\alpha$. Il'est évident que les éléments nécessaires pour calculer $\mu(\tau)\alpha$ ne sont pas connus à l'avance, et le but de la stratégie descendante est de ne pas prendre en compte des éléments à part.

L'algorithme construit la table partielle par simulation d'exécution de procédure tau en α. La simulation est directe jusqu'au moment où l'on doit appliquer la fonction f pour un paramètre β. A partir de ce moment-là, la simulation de l'algorithme est suspendue et la sous-simulation dans β est lancée. Au retour de la sous-simulation, l'algorithme modifie la table partielle et clôture la simulation initiale en utilisant dans les deux cas le résultat de sous-simulation. Dans le cas où f est appliqué à α, alors, au lieu de lancer la sous-simulation, on utilise la valeur courante de α trouvée dans la table partielle. L'algorithme relance la sous-simulation dans β si la valeur de α est redéfinie.

On peut établir une correspondance entre une table créée et la séquence des fonctions partielles $f_0, ..., f_n$, telles que

$f_0 \leq f_1 ... \leq \mu(\tau)$.

2.7.3. Table partielle

Le but de l'algorithme est de créer une fonction partielle représentée par sa table. Les tables des fonctions partielles sont appelées par pt.

Les opérations sur les tables partielles sont

◊ d'introduire un nouveau élément dans pt,

Extend_pt(pt,α)

Pre: $\alpha \notin dom(pt)$

Post: $pt=pt^{\circ} \cup \{\alpha, \beta\}$ où β= lub $\{pt^{\circ}(\alpha'): \alpha' \leq \alpha$ et $\alpha' \in dom(pt^{\circ})\}$

◊ de modifier pt avec le résultat β pour α.

Adjust_pt(pt,α,β)

Pre: $\alpha \in dom(pt)$

Post: $pt=pt_1 \cup pt_2$ où $pt_1=pt^{\circ} \setminus \{\alpha,' \beta\}' \in pt^{\circ} | \alpha' \geq \alpha\}$ et

$pt_2= \{(\alpha',$ lub $\{pt^{\circ}(\alpha'), \beta\}) | \alpha' \geq \alpha$ & $\alpha' \in dom(pt^{\circ})\}$.

2.7.4. Le graphe de dépendance

L'avantage majeur de l'algorithme général est qu'il élimine des calculs redondants. Par exemple, au cours de simulation associée avec la valeur β, on n'utilise pas le paramètre fonctionnel; alors, il doit être calculé seulement une fois. L'appel de β a pour effet de renvoyer la valeur enregistrée dans la table partielle.

De même manière, si pendant la sous-simulation associée avec la valeur de β, on utilise les paramètres fonctionnels appliqués aux les valeurs $\alpha_1, ..., \alpha_n$ et seulement à elles, on doit les recalculer quand au moins une des valeurs de $\alpha_1, ..., \alpha_n$ est changée dans la table. Pour recevoir cet effet algorithme, on utilise des dépendances dynamiques.

◊ le graphe de dépendance est une fonction partielle de A à γ(A), l'ensemble des éléments $(\alpha, \{\beta_1, ..., \beta_n\})$ tel que $(\alpha, S) \in dg$ & $(d,S') \in dg \Rightarrow S=S'$

◊ le domaine de dg , dom(dg) est un ensemble de α tel que $(\alpha, S) \in$ dg pour certaine S

◊ le codomaine de dg, codom(dg) est un ensemble de tous β tel que $\exists (\alpha, S) \in$ dg & $\beta \in S$, si $\alpha \in$ dom(dg), dg(α) est l'ensemble de S tel que $(\alpha, S) \in$ dg.

De façon intuitive, dg(α) représente tous les éléments qui dépendent de α.

Les opérations sur le graphe de dépendance sont les suivantes:

◊ enlever un élément,

Remove_dg(dg, $\{\beta_1, ..., \beta_n\}$)

Effet: on enlève un élément de graphe de dépendance avec codom=$\{\beta_1, ..., \beta_n\}$

◊ ajouter un élément,

Extend_dg(dg, α)

Effet: on ajoute un élément ,domain= α et

codom=$\emptyset$

◊ modifier le graphe de dépendance:

Add_dg(dg, α, β)

Effet:on ajoute à codom α un élément nouveau β.

2.7.5. Algorithme

L'algorithme général de calcul de point fixe a la forme suivante:

```
procedure compute fixpoint( in α : A; out pt, dg);
        var ics: set of A;
  procedure reapeat_computation(in α: A);
      var β :  A' ;
      begin
         if α∉ dom(dg) ∪ ics then
    begin
    if α ∉ dom(pt) then Extend_pt(pt, α);
    ics:= ics ∪ {α};
    repeat
        Extend_dg(dg, α);
        β:= tau( pretended_f, α);
        Remove_dg( dg, modified)
    until α∈ dom(dg);
    ics:= ics \ {α}
    end
  end;
function pretended_f( in β,α :A) :A;
  begin
    repeat _computation(β);
    if α∈ dom(dg) then Add_dg(dg, α, β);
    pretended_f:= pt(β)
  end;
begin
    pt:= 0;
    dg:=0;
    ics:=0;
    repeat_computation(α)
end.
```

La procédure générale *compute_fixpoint* reçoit à l'entrée un élément α de A et renvoit deux sorties :

◊ la table partielle pt,

◊ le graphe de dépendances dg.

Procédure tau est un objet global, elle est déterminée par l'utilisateur. La table partielle renvoyée comme résultat doit satisfaire à la condition pt(α)= μ(τ)α pour α∈ dom(dg), τ est une

transformation déterminée par tau. La procédure *compute_fixpoint* utilise un ensemble auxiliaire ics, ics représente l'ensemble des simulations initialisées. Le corps de procédure initialise des ensembles à vide et appelle la procédure *repeat_computation.*

Procédure *repeat_computation* prend une place centrale de l'algorithme. Elle commence par contrôler si le paramètre α est dans le domaine de graphe de dépendances ou dans l'ensemble de calculs déjà initialisés. Dans le premier cas, on n'a pas intérêt à exécuter des instructions restantes, parce qu'aucun élément pouvant influencer la valeur de α n'était changé. Dans le deuxième cas, le sous-calcul pour α était déjà lancé. Dans tous les autres cas, la procédure ajoute un nouvel élément dans la table partielle si nécessaire, et modifie ics en ajoutant α. Ensuite, elle entre dans la boucle , qui calcule le point fixe local pour α étant donné les valeurs des éléments suspendues dans ics.

Le point fixe est obtenu par exécution successive de programme tau et le sous-calcul se termine quand α est dans le domaine de graphe de dépendance(cela signifie que l'on ne gagne plus aucune information nouvelle par itération additionnelle). Chaque itération calcule la valeur de β utilisée consécutivement pour modifier la table partielle. Le graphe de dépendance est modifié avant l'appel; on ajoute α , et peut être modifié après l'appel en retirant des éléments qui dépendent des éléments en modified.

L'appel de procédure tau utilise la fonction *pretended_f* comme argument fonctionnel. Cette fonction reçoit deux paramètres: β est un argument à appliquer, et α est une valeur qui a besoin de valeur de β.

Pour obtenir la valeur de β, la fonction appelle de façon récursive la procédure repeat_computation pour β. Ensuite, elle modifie le graphe de dépendance si α est encore dans le domaine du graphe de dépendance; sinon, il ne faut pas faire des itérations supplémentaires et on ne modifie pas le graphe de dépendance. La procédure renvoit la valeur pt(β).

On n'entre pas dans les détails de preuve du fait que l'algorithme général est correct ni de terminaison d'algorithme. Pour plus de détails, le lecteur est renvoyé à [12]. Remarquons seulement que l'algorithme peut être appliqué au domaine infini également, et même sous certaines conditions aux transformations non-monotones.

2.7.6. Complexité

La complexité asymptotique (le nombre d'itérations dans la boucle *repeat_computation*) était analysée pour quelques classes de programmes. Dans le "pire" des cas, la complexité est égale à $O(n^2hs^2)$, où n est la taille d'un programme à analyser; h est la hauteur de cpo **A,** et s est la taille de cpo **A.**

Sous les conditions raisonnables satisfaites par beaucoup de programmes, la complexité peut être réduite à $O(nhs^2)$, O(nhs) et même O(nh).

2.7.7. Instantiation de l'algorithme

L'algorithme général de calcul de point fixe était utilisé comme une base pour développer quelques algorithmes pour l'interprétation de Prolog. La méthode générale consiste à instantaner la procédure tau pour l'implémentation d'une sémantique abstraite adéquate.
Par exemple, l'algorithme décrit dans [11,13] est dérivé d'un algorithme universel où la transformation τ est une simple sémantique d'entrée-sortie basée sur la sémantique opérationnelle standard de Prolog.

2.8. Algorithme local de calcul de point fixe

L'algorithme général calcule le point fixe pour une transformation quelconque dans un sens large, dès que la transformation répond à la propriété de monotonie.
Dans notre travail, on présente une instantanation de l'algorithme général de calcul de point fixe, notamment pour un système monotone d'équations. Il est local dans le sens qu'il localise le calcul autour d'une variable ou d'un groupe des variables voisines nécessaire pour effectuer les calculs.
Le code-source pour cet algorithme est fourni dans l'annexe 2.

2.8.1. Environnement

L'algorithme est valable également pour un système d'inéquations préalablement modifié selon les règles de ré-écriture (**) de section 2.2.
Nous sommes dans l'environnement suivant:

- ◊ transformation τ est répresentée par un ensemble fini d'équations
  $x_i=f_i$ $(1\le i\le n)$
  $x_1, ..., x_n$ sont des variables différentes;
- ◊ les variables $x_1, ..., x_n$ sont rangées dans le treillis d'hauteur fini ;
- ◊ $f_1, ..., f_n$ sont des expressions monotones, peuvent également contenir $x_1, ..., x_n$.

2.8.2. Présentation

L'algorithme calcule le plus petit point fixe pour α, qui est défini par l'utilisateur dans le mode interactif.
L'utilisateur définit lui-même la transformation τ en introduisant le fichier de texte qui doit contenir le système d'inéquations pour lequel il voudrait calculer le plus petit point fixe.

Sans entrer dans les détails de la structure des données et de l'algorithme-même, il faut remarquer qu'il s'inspire de l'algorithme général de calcul de point fixe.
On travaille avec des tableaux:

- ◊ pt (la table des valeurs des fonctions partielles);
- ◊ ics (indique les simulations déjà en route);

◊ dg (le tableau de graphe de dépendance, le numéro dans le tableau dg indique la position de variable dans le tableau des variables et répresente le domaine du graphe de dépendances, un élément du graphe de dépendances correspond au codomaine du graphe de dépendance);

◊ optimal (le tableau de type entier est mis en correspondance avec le tableau des variables utilisées par le système d'inéquations. La présence de 1 dans le tableau optimal signifie que la valeur d'une variable correspondante est "bonne " et qu'il n'est pas besoin de la raffiner. La présence de 0 indique qu'il faut lancer la nouvelle simulation pour cette variable afin d'améliorer la valeur par le résultat des calculs).

L'algorithme local est construit à la base de deux sortes de dépendances: dépendance statique et dépendance dynamique.

Pour chaque variable rencontrée dans la partie gauche des inéquations, on construit la chaîne de dépendance statique suite[x], où on trouve les variables de la partie droite de ces inéquations.

Pour trouver la valeur d'une variable qui figure dans la tête d'inéquation, on est obligé de lancer les sous- calculs pour chaque variable de la partie droite. Mais cela ne signifie pas chaque fois qu'on relance la procédure entière. Si la valeur est optimale, la sous- simulation prévoit tout simplement la recherche de la valeur dans la table pt[x]. L'algorithme local garde l'avantage de l'algorithme général de minimiser le nombre de calculs nécessaires pour trouver le plus petit point fixe.

La procédure 'calcul' fait les calculs de valeurs proprement dites; elle reçoit comme paramètres:

◊ pile (pile des opérandes et opérateurs à manipuler);

◊ borne (la fourchette qui indique la première et la dernière variable dans la pile, permet de trouver la bonne inéquation des calculs appropriés).

La fonction calcul renvoit en sortant:

◊ résultat (la valeur d'une variable qui se trouve dans la partie gauche d'inéquation).

A la place de trois procédures utilisées par l'algorithme général pour organiser la structure des dépendances dynamiques, on utilise deux procédures:

◊ *CreerDg(y, x, dg)* (ici le domaine est égale à y, on ajoute au codomaine correspondant du graphe de dépendances la variable numéro x);

◊ *MettreNul(x, dg)* (on met nul à codomaine pour le domaine égale à x, la boucle est organisée pour mettre à nil tous les codomaines qui correspondent aux domaines égaux aux indices des variables trouvées dans les codomaines précédemment annulés).

L'algorithme local de calcul de point fixe fait les calculs strictement nécessaires pour trouver le plus petit point fixe pour variable α, le nombre de sous-calculs faits par l'algorithme dépendant de la nature de transformation τ. Il peut arriver que l'on obtienne comme résultat d'un algorithme les points fixes calculés pour toutes les variables d'un système d'inéquations ou d'une grande partie d'entre eux. Le nombre des simulations effectuées dépend également de la position dans le système d'inéquations de variable pour lequel on veut faire des calculs.

La position centrale de l'algorithme local prend la procédure *eval*.

La procédure eval reçoit en entrée

- ◊ x (le numéro de la variable locale pour calculer le point fixe),

  et renvoit comme résultat
- ◊ dg (le graphe de dépendance qui est changée au cours des évaluations),
- ◊ pt (la table des fonctions partielles calculées),
- ◊ optimal (la table remplit par 1 dans les positions correspondant aux points fixes calculés par l'algorithme).

Pour plus de détails sur le schéma de la procédure *eval*, des procédures *CreerDg*, *MettreNul* et des autres procédures et fonctions, veuillez vous référer au chapitre 4 "Programmation de l'algorithme".

Chapitre 3. SYNTAXE DU LANGAGE SYSTINEQ

3.1. Constructions du langage SYSTINEQ

Tout programme SYTINEQ est défini comme étant une certaine suite finie de symboles de base.
L'ensemble des symboles de base est défini par une grammaire BNF dans l'annexe 1.
Parmi toutes les suites de symboles, on reconnaît les constructions du langage.
Les différentes constructions peuvent être réparties dans les catégories suivantes:

- déclarations des variables et des tableaux;
- expressions de désignation: constructions permettant de désigner une variable particulière ;
- expressions: calcul d'une variable;
- programme: spécification d'un traitement complet à effectuer sur un fichier d'entrée.

3.2. Syntaxe

On définit la syntaxe des constructions du langage au moyen de grammaires BNF:
< programme>::= <en tete de programme><bloc>
<en tete de programme>::= procédure<nom de procédure>;
<nom de programme>::= <identificateur>
<bloc>::= <déclaration des variables et des tableaux><suite d'inéqutions>
<déclaration des variables et des tableaux>::= <vide>| var<liste des déclarations des variables ou des tableaux>
<liste des déclarations des variables ou des tableaux>::=< déclaration des variables ou des tableaux><liste des déclarations des variables ou des tableaux>
<liste des déclarations des variables ou des tableaux>::= <déclaration de variables simples>| <déclaration des tableaux>
<déclaration des variables simples>::= <nom de variable>:<type>| <nom de variable>, <déclaration des variables simples>
<nom de variable>::= <identificateur>
<déclaration des tableaux>::= <nom de tableau>:array[<entier>..<entier>] of <type>| <nom de tableau>, <déclaration des tableau>
<nom de tableau>::= <identificateur>
<vide>::=
<suite d'inéquation >::= begin <liste des inéquations>end

<liste des inéquations>::= <inéquation>| <liste des inéquations>; <inéquation>

<inéquation>::=<expression>

<expression>::= <expression simple>|<expression arithmétique>|<expression booléenne>

<expression simple>::= <expression de désignation>|<constante>| (<expression>)

<expression arithmétique>::=<terme>| <expression arithmétique><opérateur additif><terme>-terme

<opérateur additif>::= +|-

<terme>::= <facteur>| <terme>| <opérateur multiplicatif>

<facteur>::= <expression simple>

<opérateur multiplicatif>::= *|div|mod

<expression booléenne>::=<conjonction>| <conjonction>or<expression booléenne>

<conjonction>::= <négation>| and <conjonction>

<négation>::= <proposition atomique>| not <proposition atomique>

<proposition atomique>::= <expression arithmétique>| <expression arithmétique><opérateur relationnel><expression arithmétique>

<opérateur relationnel>::= =|<>| <| >| <=| >=

Exemples:

◊ déclaration de variables simples et de tableaux

```
var
i, j, premier: integer;
x              : array[1..50]of integer;
```

◊ expressions de désignation

```
x
a[x]
```

◊ constantes

```
0
435
'b'
```

◊ expressions paranthésées

```
(x)
((((x))))
```

```
(((((x+y)*a)-b)+")div2)*0)
```

◊ facteurs

```
x
300
(a+b*c=d div i)
```

◊ termes

```
x
x*300
x*300 div (a+b*c=d div i)
```

◊ expressions arithmétiques

```
x
x*300
a+b*c=d div i
a+(b*c)-(d div i)
```

◊ expressions atomiques

```
p
existe(x, a)
true
g[i]<= t[k]
```

◊ négations

```
existe(x, a)
not existe(x, a)
```

◊ conjonctions

```
existe(x,a)
p and not existe(x, a)
not p and existe(x,a) and g[i]<=t[k]
```

◊ expressions booléennes

```
existe(x, a)
p or not existe(x, a) and g[i]<= t[k]
```

◊ inéquation

```
b<= x div y
```

◊ suite d'inéquations

```
begin  a=x+y; b<= x div y; c>=x end
```

◊ programme

```
program essai2;
var
x1,x2,x3,p,a,b,c:  integer;
     begin

          x1>=max(a,1);
          x2>=x1*b;
          p>=x2+c;
          a>=0;
          b>=1;
          x3>=a*c;
          c=>b+x3;
          end.
          ?
```

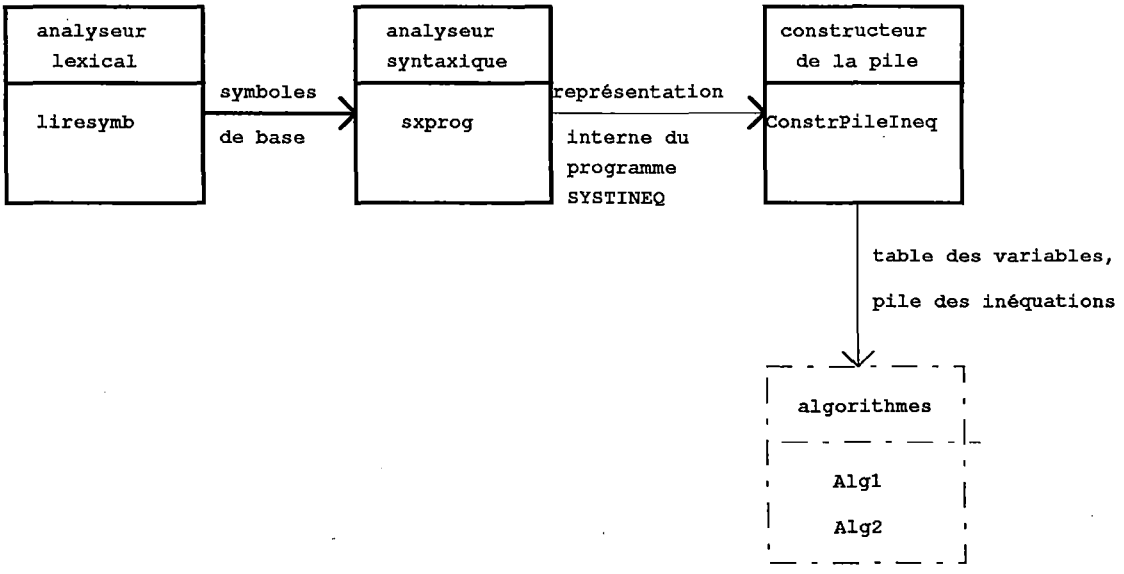
Chapitre 4. IMPLEMENTATION DE L'INTERPRETEUR

Ce chapitre présente l'interpréteur permettant de construire, à partir d'un système d'inéquations, une pile toute prête d'opérandes et d'opérateurs fonctionnels. Cette pile servira de base au développement des algorithmes Bottom-Up et Top-Down.
Le point de départ de l'interpréteur est un fichier de texte qui contient la description d'un système d'inéquations, selon la syntaxe du langage SYSTINEQ définie au chapitre précédent. L'interpréteur se divise en trois parties :

- l'analyseur lexical,
- l'analyseur syntaxique,
- le constructeur de la pile.

4.1. Architecture globale

Le schéma ci-dessous montre le découpage de l'interpréteur en modules et l'échange de données entre modules :



L'interpréteur du langage SYSTINEQ est un programme Pascal, utilisant les fichiers standards d'entrée.
Si l'on fournit comme données à l'interpréteur un programme SYSTINEQ, il fournit comme résultat le tableau des variables utilisées dans la déclaration des variables du programme et une pile d'inéquations, si cette exécution est bien déterminée et se termine. Si l'on exécute l'interpréteur avec, pour données, une chaîne de caractères dont aucun préfixe n'est la représentation d'un

programme SYSTINEQ suivie d'un ?, cette exécution se termine après l'impression d'au moins un message d'erreur à l'écran.

Le module "Algorithme" figure dans la représentation schématique de l'interpréteur pour en respecter la sémantique. Dans notre réalisation, aucun des trois modules "Analyseur lexical", "Analyseur syntaxique", "Constructeur de pile" n'effectue le calcul des valeurs. Ils jouent un rôle préparatoire, chacun préparant tour à tour un programme écrit en suivant la syntaxe du langage SYSTINEQ pour confier ensuite l'évaluation au module "Algorithme". L'interpréteur est dédicacé au module " Algorithme ", qui contient deux méthodes de calcul, appelées respectivement Bottom-Up et Top-Down.
Le mécanisme de calcul et d'évaluation des programmes SYSTINEQ est décrit au Chapitre 5 "Implémentation des algorithmes".

4.2. Analyseur lexical

On appelle analyse lexicale le traitement consistant à reconstruire une suite de symboles de base, S, à partir d'une représentation externe, α , de S.
L'analyseur lexical fournit à l'analyseur syntaxique la suite de base. Il est beaucoup plus facile à l'analyseur syntaxique de manipuler les symboles de base que les caractères de représentation externe du programme.

4.2.1. Représentation externe

La représentation externe d'un système d'inéquations est la représentation de ce système en concaténant les symboles de base, tout en respectant les règles suivantes:
- les lettres minuscules qui figurent dans les identificateurs sont remplacées par les lettres majuscules correspondantes;
- les symboles spéciaux sont remplacés par les suites correspondantes de lettres majuscules;
- un nombre quelconque de caractères "espace" peut être inséré entre deux symboles de base du système d'inéquations;
- il faut insérer au moins un caractère "espace" partout où la concaténation directe de deux symboles peut créer une ambiguïté.

L'analyse du texte consiste à reconstituer une suite de symboles de base à partir d'une représentation externe de cette suite de symboles de base.

4.2.2. Représentation interne d'un symbole de base

Les symboles de base sont représentés de façon interne par des valeurs du type Pascal " bsymb " :

```
type
chaîne = array [1..8] of char;
tbsymb= (identificateur, caractère, booléen, entier, specsymb,
nosymb, errsymb);
bsymb= record
                {représentation interne d'un symbole de base }
       ibsymb: tbsymb;
                {indique de quel genre de symbole de base il
s'agit }
       case ibsymb of
                {précise l'identité du symbole de base}
       identificateur,specsymb :(cval:chaîne);
       caractère,booléen,entier:(consval:integer);
       nosymb, errsymb         : ( )
       end;
```

où

•*ibsymb* est l'indicateur;

• *cval* est la représentation externe d'un symbole de base cadrée à gauche, et éventuellement tronquée à ses 8 premiers caractères;

• si s est une constante de la forme 'c' où c est un des caractères énumérés dans l'ensemble des valeurs de type caractère, le rang i de c, dans cette énumération, est placé dans consval;

• si s est une constante booléenne, consval = 0 si s est false, = 1 si s est true;

• si s est une constante entière, le nombre entier i dont s est la représentation décimale est placé dans *consval*.

Exemples:

◊ Soit s est l'identificateur *toto*, alors on a:

identificateur	TOTO
ibsymb	*cval*

◊ Soit s est la constante 'c', alors on a:

caractère	4
ibsymb	*cval*

◊ Soit s est la constante 'true', alors on a:

bolean	0
ibsymb	*cval*

◊ Soit s est la constante '5240', alors on a:

entier	5240
ibsymb	*cval*

4.2.3. Spécification de l'analyseur lexical

L'analyseur lexical est une procédure écrite en Pascal, de nom *liresymb*, sans paramètre, qui lit le symbole de base suivant dans le fichier standard d'entrée (input).

Nom: ♦ liresymb;

Structure des données:

◊ une variable *symb* de type bsymb;

◊ un tableau *tabcar* de type array[1..nbrcar] of char,
où nbrcar : constante ayant pour valeur le nombre d'éléments de l'ensemble des valeurs de type caractère;

Remarque : tabcar[i] doit avoir pour valeur le ième caractère de l'énumération de cet ensemble lors d'un appel de cette procédure.

Effet:

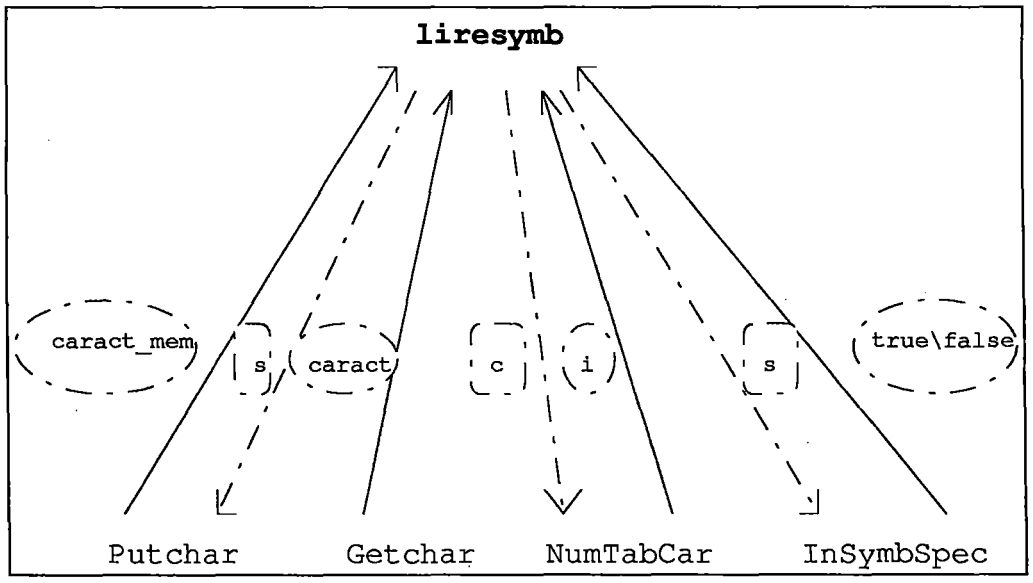
Soit α la suite de caractères restant à lire sur le fichier d'entrée.
α se décompose d'une seule façon sous la forme $\alpha'\beta$
où α' est une suite, peut-être vide, d'espaces
β est une suite, peut-être vide, de caractères, ne commençant pas par un espace.

•S'il existe un plus grand préfixe σ de β, tel que σ est la représentation externe d'un symbole de base s, [β est alors de la forme $\sigma\gamma$; l'appel de procédure "liresymb" affecte à symb la représentation interne de s et laisse le fichier d'entrée dans l'état γ. [γ est alors la chaîne des caractères restant à lire sur le fichier d'entrée]

•Si β est de la forme ?β', l'appel de procédure "liresymb" affecte à symb.ibsymb la valeur nosymb et laisse le fichier d'entrée dans l'état β'.

•Sinon, l'appel de procédure "liresymb" affecte à symb.ibsymb la valeur errsymb et laisse le fichier d'entrée dans un état indéterminé.

4.2.4. Graphe d'appel de l'analyseur lexical



La procédure *liresymb* appelle:

♥ *Getchar*, la fonction

•renvoie à liresymb *caract* mis en majuscule;

♥ *Putchar*, la procédure

- reçoit une variable *s* de type char,
- place le caractère dans *caract_memoire*;

♥ *NumTabCar*, la procédure

- reçoit *c* de type char,
- renvoie *i* le rang de *c* dans l'énumération formée de valeurs de type char(*tabcar*);

♥ *InSymbSpec*, la fonction

- reçoit la variable *s* de type chaîne,
- indique à *liresymb* si la chaîne se trouve dans le tableau *SymbSpec* ou pas.

4.3. Analyse syntaxique

4.3.1. Résumé des fonctionnalités

L'analyse syntaxique consiste à vérifier que le fichier standard d'entrée contient bien la représentation externe d'un programme SYSTINEQ, et à créer une représentation interne de ce programme.
Si la chaîne de caractères à analyser n'est pas la représentation externe d'un programme SYSTINEQ, l'analyseur syntaxique doit signaler ce fait par l'impression d'un message d'erreur.

4.3.2. Principes de l'analyse syntaxique

Déroulement normal

Pour construire l'analyseur, nous allons écrire un ensemble de fonctions Pascal capables de faire l'analyse syntaxique des constructions du langage SYSTINEQ appartenant à une catégorie syntaxique bien déterminée. Chacune de ces fonctions sera appelée dans un environnement tel, que la suite des symboles à traiter doive commencer par une construction de la catégorie syntaxique correspondante si le programme SYSTINEQ à analyser est correct.
Il est possible que plusieurs préfixes de la suite de symboles à traiter, appartiennent à une catégorie syntaxique considérée. En fait, le préfixe qu'on aura intérêt à traiter, sera toujours le plus long, et sera toujours cat-maximal, quelle que soit la circonstance de l'appel (si le programme à analyser est "syntaxiquement correct")

Terminaison anormale

On décidera d'arrêter prématurément l'exécution de l'interpréteur lorsque l'analyse syntaxique est terminée, si une erreur au moins a été détectée.

L'exécution d'un appel quelconque de fonction ou de procédure de l'interpréteur se termine anormalement lorsque cette exécution déclenche l'arrêt immédiat de l'interpréteur au lieu de se terminer par un retour au point d'appel.

Représentation interne d'un programme SYSTINEQ sous forme d'arbre

Nous allons définir des règles de traduction permettant de représenter les principales constructions du langage SYSTINEQ par des structures de données Pascal:

Représentation interne[3] des expressions, des inéquations, des déclarations des variables et du programme

Chaque construction de ce type sera représentée par une zone générée dynamiquement (≡ variable dynamique). Elles sont toutes du type "constr" :

```
type tconstr=(fonction,appel,simvar,indvar,unop,binop,
         const,affect,cond1,cond2,debut,vari,tab,val,prog);
valtype=caractere..entier;
constr=record
          iconstr:tconstr;
          case iconstr of
             prog:
                (pnom:chaine;
                 lvar:^lconstr;
                 corps:^constr);
             fonction:
                (fnom:chaine;
                 leff:^lconstr);
             simvar:
                (vnom:chaine);
             indvar:
                (tnom:chaine;
                 indice:^constr);
             unop:
                (uop:operateur;
```

[3] Désormais, nous utiliserons le terme de 'représentation interne' pour 'représentation interne sous forme d'arbre'.

```
                oper:^constr);
            binop:
                (bop:operateur;
                 oper1,oper2:^constr);
            const:
                (constype:valtype;
                 consval:integer);
            cond1,cond2:
                (bexpr1,instr1,instr2:^constr);
            debut:
                (lineq:^lconstr);
            vari,val:
                (vartype:valtype;
                 lvnom:^lident);
            tab:
                (tabtype:valtype;
                 ltnom:^lident;
                 db,fb:integer)
        end;
operateur=(fois,quotient,reste,modulo,plus,moins,egal,different,

inferieur,superieur,infeq,supeq,et,non,ou,maximum,minimum,compar);
```

Représentation interne des déclarations des programmes

Soit la déclaration de programme

```
proqram p;
    var decv1;...;decvm;
    cins
```

Cette déclaration sera représentée par une variable dynamique de type "constr" ayant le contenu suivant:

prog	p	↑(decv$_1$;...;decv$_m$)	↑cins
iconstr	pnom	lvar	corps

Représentation interne des appels de fonctions et des expressions

fonction	*identificateur*	^(l_1;...;l_n)
iconstr	fnom	leff

simvar	*identificateur*
iconstr	vnom

indvar	*identificateur*	^*expression arithmétique*
iconstr	tnom	indice

const	entier	*valeur de la constante entière*

const	caractère	*valeur de la constante caractère*

const	booleen	*valeur de la constante booléenne*
iconstr	constype	consval

unop	ω'	^l_1
iconstr	uop	oper

binop	ω'	^l_1	^l_2
iconstr	bop	oper1	oper2

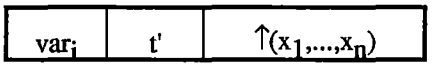
Note : Une expression entre parenthèses se représente comme l'expression obtenue en supprimant les deux parenthèses extérieures.

debut	^(inq_1;...;inq_n)
iconstr	lineq

Représentation interne des déclarations de variables et de tableaux, des groupes d'arguments, et des paramètres formels

La construction $x_1,...,x_n$: t se présente sous une des formes ci-dessous :

dans le cas de variables simples



iconstr vartype lvnom

dans le cas d'un groupe d'arguments formels

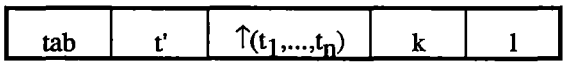
iconstr vartype lvnom

Les constructions

var $x_1,...,x_n$: t {groupe de paramètres formels}

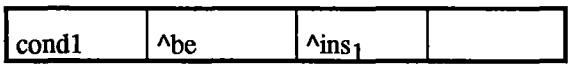
var $t_1,...,t_n$: array[k..l] of t {groupe de paramètres formels de type tableau}

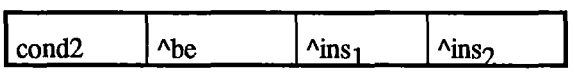
se représentent de la façon suivante :



iconstr tabtype ltnom db fb

Représentation interne des expressions conditionnelles





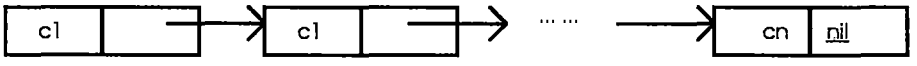
iconstr bexpr1 instr1 instr2

Représentation interne des suites de constructions

Les suites de constructions comportant un nombre quelconque d'éléments sont représentées par une suite de variables dynamiques du type "lconstr":

```
type lconstr=record
                pconstr:^constr;
                lsconstr:^lconstr
             end;
```

Soit $c_1,...,c_n$, une suite de constructions du langage SYSTINEQ (expressions, inéquations, déclarations d'arguments ou de paramètres formels, de variables ou de tableaux). Cette suite sera représentée par une liste chaînée:



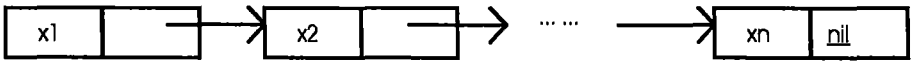
Si la suite de constructions est non vide (n>=1), $\wedge(c_1;...;c_n)$ est le pointeur vers la première cellule de la suite de cellules; sinon c'est le pointeur nil.

Représentation interne des suites d'identificateurs

Les suites d'identificateurs figurant dans les déclarations de variables ou de tableaux, et dans les groupes d'arguments et de paramètres formels d'un programme SYSTINEQ, seront représentées par une suite de variables dynamiques du type "lident" :

```
type lident=record
              pident:chaine;
              lsident:^lident
            end;
```

La suite d'identificateurs $x_1,...,x_n$ sera représentée par une liste chaînée:



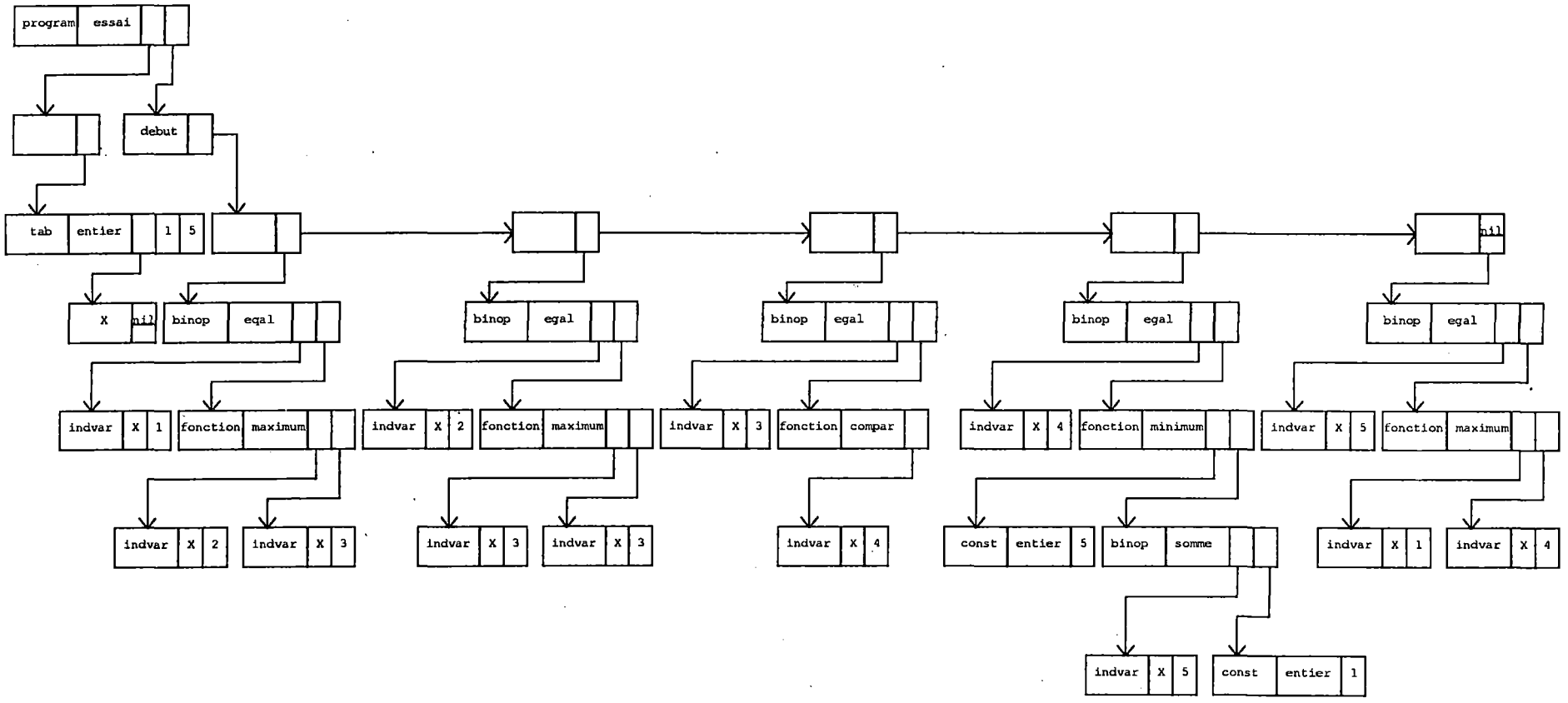
$\wedge(x_1,...,x_n)$ est le pointeur vers la première cellule.

Exemple:

- Soit, on a un programme écrit en langage SYSTINEQ

```
program essai;
var
x : array [1..5] of integer;
begin
      x[1]=max(x[2],x[3]);
      x[2]=max(x[3],x[3]);
      x[3]=comp(x[4]);
      x[4]=min(5,x[5]+1);
      x[5]=max(x[1],x[4]);
end.
?
```

- La représentation interne sous forme d'un arbre est la suivante:



Représentation d'un programme "Essai" sous forme d'un arbre

4.3.4. Implémentation de l'analyseur syntaxique

L'analyseur syntaxique est représenté par la fonction *sxprog*:

Fonction sxprog

Nom : ♦ sxprog : ty_constr;

Paramètres : ◊

Effet: • la fonction renvoie un pointeur vers la représentation interne du programme SYSINEQ analysé, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte.

Spécification de l'analyseur syntaxique

•Si la suite des caractères restant à lire dans le fichier standard d'entrée est de la forme π?, où π est une représentation externe d'un programme SYSTINEQ (soit p ce programme), la fonction sxprog lit la suite de caractères π? dans le fichier d'entrée, et renvoie le pointeur $\uparrow$p vers la représentation interne de p.

• Sinon, la fonction se termine anormalement après avoir imprimé un message d'erreur.

Impression des messages d'erreur

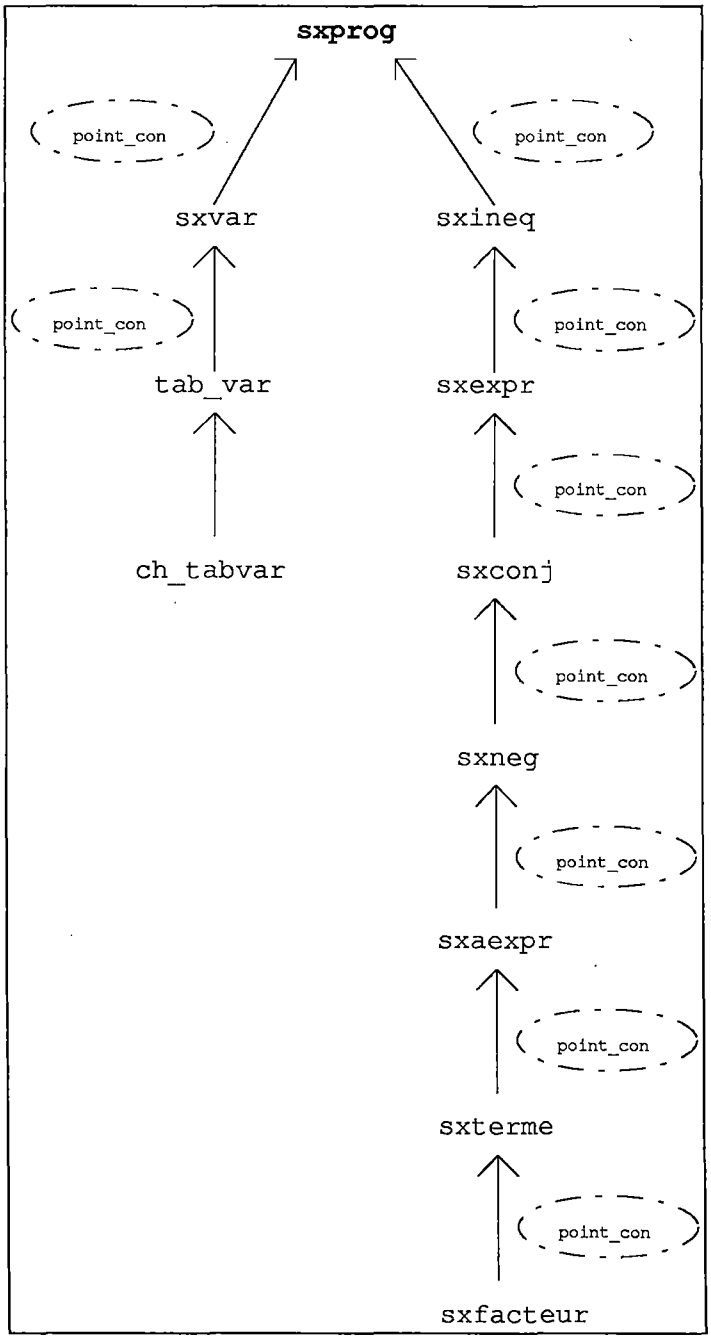
Elle s'effectue par un appel à la procédure *affichage* qui reçoit en paramètres le code de l'erreur et le numéro de ligne où l'on a repéré l'erreur.

La procédure *affichage* appelle la procédure *ecriremsg(msg:message)* où "message" est défini comme suit :

```
type message=packed array[1..4] of char.
```

Graphe d'appel

L'organisation du module "Analyseur syntaxique" peut être représentée par le schéma[4] suivant :



[4] Chaque fonction de l'analyseur syntaxique fait appel au module "Analyseur lexical", plus précisément à la procédure *liresymb* et à la procédure *affichage*.

La procédure *sxprog* appelle :

♥ *sxvar*, la fonction

•renvoie à *sxprog* une variable *point_con* , le pointeur vers la représentation interne sous forme d'un arbre des déclarations de variables;

♥ *sxineq*, la fonction

•renvoie à *sxprog* une variable *point_con* vers la représentation interne sous forme d'arbre d'une suite d'inéquations.

4.4. Constructeur de la pile

4.4.1. Résumé des fonctionnalités

La construction de la pile est le traitement qui consiste à construire, à partir de la représentation interne sous forme d'arbre du programme SYSTINEQ, la représentation interne sous forme de pile. C'est en phase de construction de la pile, que le programme SYSTINEQ est reconnu syntaxiquement correct ou non.

La construction de la pile se fait en deux étapes: la première étape consiste à construire une table des variables utilisées par le programme SYSTINEQ à partir de la représentation interne des déclarations de variables; la deuxième s'occupe de la construction de la pile proprement dite. Dans la pile, les variables sont remplacées par leur position dans la table des variables construite auparavant. Les fonctions et les opérateurs sont indiqués par les numéros correspondants trouvés dans la table des fonctions.

4.4.2. Construction d'une table des variables

La table des variables est construite par la procédure *ConstrTabVar*, qui reçoit comme paramètre une représentation interne des déclarations de variables (fournie par l'analyseur syntaxique).

Procédure ConstrTabVar

Nom: ♦ ConstrTabVar;

Paramètres: ◊ x de type ty_constr;

Structure de données:

◊ tab_var a le type(table des variables):

```
type
ty_var = record
      iconstr :tconstr;
      vnom :chaine;
```

```
      vtype :valtype;
      indice :integer
end;
ty_tab_var=array[0..NbVar] of ty_var;
```

Effet: • construit la table des variables à partir de la répresentation interne de déclarations des variables.

4.4.3. Représentation interne d'une suite d'inéquations sous forme de pile

En Pascal, la pile des inéquations a la forme d'un tableau. Chaque élément du tableau est de type:

```
type
elem_tab_ineq = record
      case iconstr : tconstr of
      simvar,indvar:
            (num_var :integer;
            vartype :valtype;
            ind :integer);
      unop,binop,fonction:
            (num_oper:integer;
            optype:valtype;
            nbr_arg :integer);
      const:
            (constype :valtype;
            consval:integer);
end;
```

Le champ *iconstr* désigne le type de construction auquel on a affaire.

iconstr	type de construction
fonction	déclaration de fonction avec le nombre d'arguments définis par *nbr_arg*
unop	expression à un opérateur et à une opérande
binop	expression à deux opérandes
simvar	expression de désignation d'une variable simple
indvar	expression de désignation d'une variable indicée
const	constante

Pour chaque construction du langage SYSTINEQ, la pile est garnie selon l'ordre postfixé.

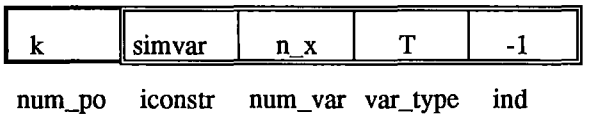
Représentation interne des expressions de désignation

Les expressions de désignation x^5 ,

où

- x est identificateur

se représentent par

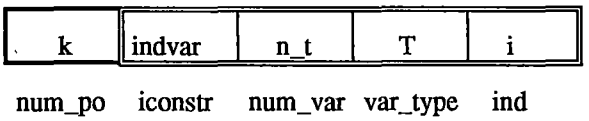


Les expressions de désignation t[i],

où

- t est identificateur et
- i est un indice

se représentent par



---

[5] Par la suite, nous montrerons la représentation interne sous forme de pile à partir des catégories de constructions. En réalité, les procédures et les fonctions du constructeur de pile utilisent une représentation interne sous forme d'arbre, mais nous utilisons l'autre forme pour simplifier.

Représentation interne des fonctions en forme de pile

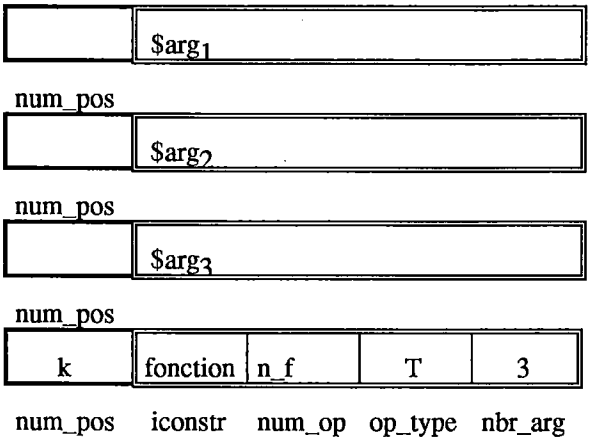
Soit la fonction

$f(arg_1, arg_2, arg_3)$

où

- f est le nom de fonction
- arg_1, arg_2, arg_3 sont des arguments de fonction

Alors la représentation interne en forme de pile est[6] :



Remarque:

La première colonne de chaque cellule, désignée par num_pos en bas de colonne, indique la position d'un élément dans la pile. Si c'est une expression différente d'une expression de désignation, alors le numéro de position est indéterminé; cette expression prend plus d'une position dans la pile. Dans notre exemple, arg_1, arg_2,arg_3 peuvent être des expressions de désignation. Dans ce cas, ils prennent chacun une position dans la pile; dans le cas contraire, ils se décomposent en plusieurs éléments de pile.

Représentation interne sous forme de pile des expressions unaires

Les expressions de la forme $\omega\ e_1$,

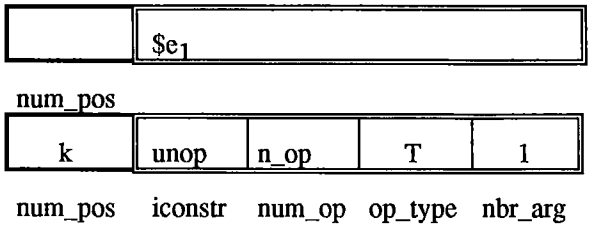
où

- ω est un opérateur unaire et
- e_1 est une expression selon les régles de syntaxe du langage SYSTINEQ

se représentent par

---

[6] Le signe $ avant une expression quelconque signifie la représentation interne sous forme de pile de cette expression.



Représentation interne sous forme de pile des expressions binaires

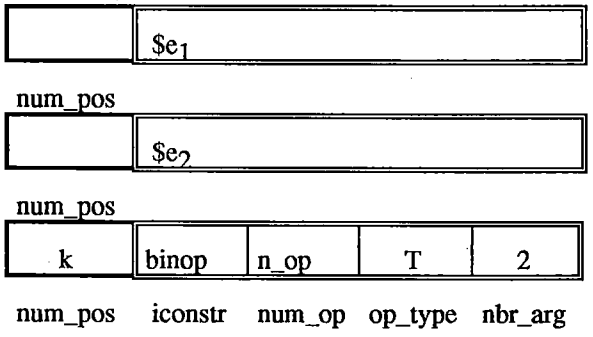
Les expressions de la forme e1 ω e_2,

où

• ω est un opérateur unaire et

• e_1 et e_2 sont des expressions selon les règles syntaxiques du langage SYSTINEQ

se représentent par



Représentation interne sous forme de pile d'une inéquation

Une inéquation peut être considéréé comme une expression binaire avec comme première opérande une variable simple ou indicée, et comme deuxième opérande une expression quelconque correspondant à la syntaxe du langage SYSTINEQ.

L'inéquation

$v_{gauche} = e_{droite}$

où

• = est un opérateur binaire

• v_{gauche} est la partie gauche de l'inéquation représentée par une variable simple ou indicée et

• e_{droite} est la partie droite de l'inéquation, représentée par une expression tout en respectant le syntaxe du langage SYSTINEQ

se représentent par

	v_{gauche}

num_pos

	e_{droite}

num_pos

k	binop	15	T	2

num_pos iconstr num_op op_type nbr_arg

Pour clarifier la représentation interne sous forme de pile d'une inéquation, prenons un exemple concret :

L'inéquation

x[i]≤max(x[i+1], x[n-1]),

où

- ≤ est un opérateur binaire
- x[i] est la partie gauche de l'inéquation représentée par une variable indicée et
- max(x[i+1], x[n-1]) est la partie droite de l'inéquation, représentée par une fonction à deux arguments

se représente par

k	indvar	i	T	i

num_pos iconstr num_va var_type ind

k+1	indvar	i+1	T	i+1

num_pos iconstr num_va var_type ind

k+2	indvar	n-1	T	n-1

num_pos iconstr num_va var_type ind

k+3	fonction	15	T	2

num_pos iconstr num_op op_type nbr_arg

k+4	binop	10	boolean	2

num_pos iconstr num_op op_type nbr_arg

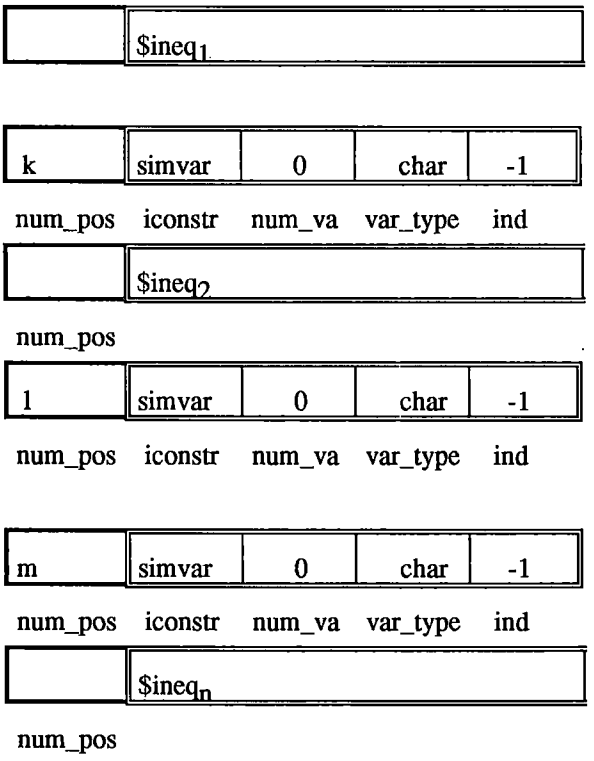
*Remarque:*Le nombre 15 est le numéro de position de fonction 'max' dans la table des fonctions; le nombre 10 correspond à la position de l'opérateur '≤' dans la table des fonctions.

Représentation interne sous forme de pile des suites d'inéquations

Chaque inéquation dans la pile des inéquations est séparée par le caractère '!', qui porte le numéro 0 dans la table des variables.

Soit $ineq_1$,$ineq_2$, ..., $ineq_n$, une suite d'inéquations du langage SYSTINEQ.

Cette suite se représente comme suit:



Exemple:

La représentation interne sous forme de pile pour le programme "Essai" (voir la représentation interne sous forme d'arbre à la page 46), est la suivante:

1	indvar	x	integer	1
num_pos	iconstr	vnom	vtype	indice
2	indvar	x	integer	2
num_pos	iconstr	vnom	vtype	indice
3	indvar	x	integer	3
num_pos	iconstr	vnom	vtype	indice
4	indvar	x	integer	4
num_pos	iconstr	vnom	vtype	indice
5	indvar	x	integer	5
num_pos	iconstr	vnom	vtype	indice

Table des variables du programme "Essai"

1	indvar	1	integer	1
num_pos	iconstr	num_var	var_type	ind
2	indvar	2	integer	2
num_pos	iconstr	num_var	var_type	ind
3	indvar	3	integer	3
num_pos	iconstr	num_var	var_type	ind
4	fonction	15	integer	2
num_pos	iconstr	num_op	op_type	nbr_arg
5	binop	6	boolean	2
num_pos	iconstr	num_op_o	op_type	nbr_arg
6	simvar	0	char	-1
num_pos	iconstr	num_var	var_type	ind
7	indvar	2	integer	2
num_pos	iconstr	num_var	var_type	ind
8	indvar	3	integer	3
num_pos	iconstr	num_var	var_type	ind
9	indvar	3	integer	3
num_pos	iconstr	num_var	var_type	ind
10	fonction	15	integer	2
num_pos	iconstr	num_op	op_type	nbr_arg

11	binop	6	boolean	2

num_pos	iconstr	num_opu	op_type	nbr_arg
12	simvar	0	char	-1

num_pos	iconstr	num_var	var_type	ind
13	indvar	3	integer	3

num_pos	iconstr	num_var	var_type	ind
14	indvar	4	integer	4

num_pos	iconstr	num_var	var_type	ind
15	fonction	17	integer	1

num_pos	iconstr	num_op	op_type	nbr_arg
16	binop	6	boolean	2

num_pos	iconstr	num_opu	op_type	nbr_arg
17	simvar	0	char	-1

num_pos	iconstr	num_var	var_type	ind
18	indvar	4	integer	4

num_pos	iconstr	num_var	var_type	ind
19	const	integer	5	

num_pos	iconstr	constype	consval	
20	indvar	5	integer	5

num_pos	iconstr	num_var	var_type	ind
21	const	integer	1	

num_pos	iconstr	constype	consval	
22	fonction	4	integer	2

num_pos	iconstr	num_op	op_type	nbr_arg
23	fonction	16	boolean	2

num_pos	iconstr	num_opop	op_type	nbr_arg
24	binop	6	boolean	2

num_pos	iconstr	num_opu	op_type	nbr_arg
25	simvar	0	char	-1

num_pos	iconstr	num_var	var_type	ind
26	indvar	5	integer	5

num_pos	iconstr	num_var	var_type	ind
27	indvar	1	integer	1
num_pos	iconstr	num_var	var_type	ind
28	indvar	4	integer	4
num_pos	iconstr	num_var	var_type	ind
29	fonction	15	integer	2
num_pos	iconstr	num_op	op_type	nbr_arg
30	binop	6	boolean	2
num_pos	iconstr	num_op	op_type	nbr_arg

Représentation interne d'un programme "Essai" sous forme de pile.

4.4.4. Construction de la pile des inéquations

La construction d'une partie de la pile correspondant à une inéquation est réalisée par la procédure *PileIneq*.

Procédure PileIneq

Nom: ♦ PileIneq;

Paramétres: ◊ n de type integer {réprésente la position du début d'un segment correspondant à l'inéquation} ;
◊ x de type ty_constr

Effet: • renvoie une partie de la pile d'inéquations composées des opérateurs et des opérandes d'une inéquation du système,
• renvoie la valeur *num* indiquant la position de fin d'une inéquation dans la pile.

La pile complète est construite par la procédure *ConstrPileIneq*:

Procédure ConstrPileIneq

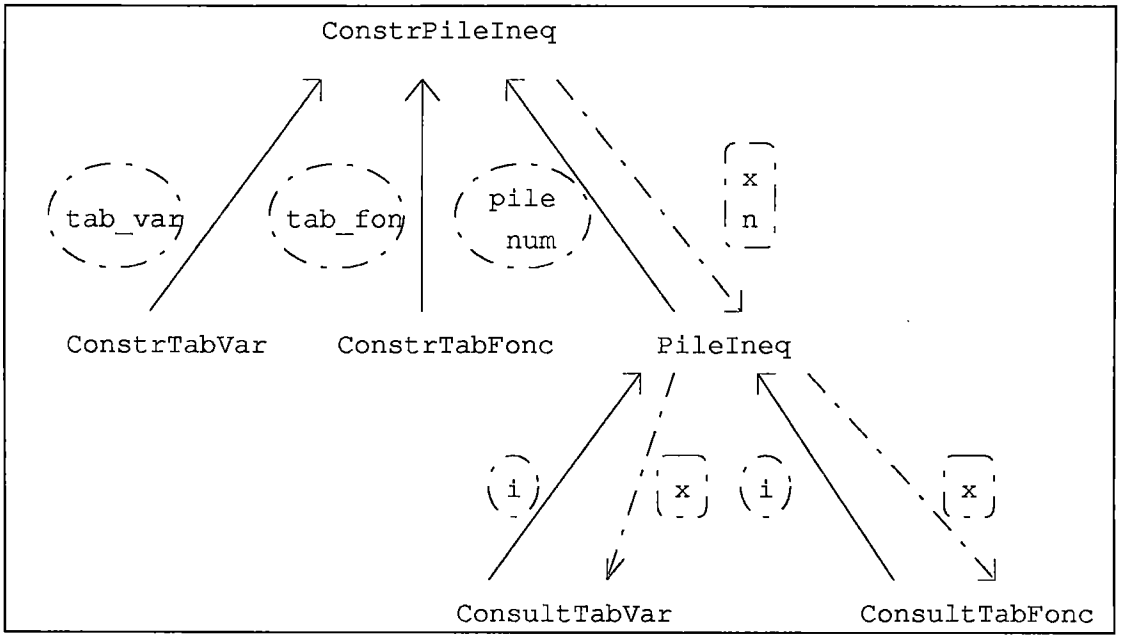
Nom: ♦ ConstrPileIneq;

Paramétres: ◊ *prg* de type ty_constr;

Effet: • renvoie la pile des inéquations composées des opérateurs et des opérandes d'un système d'inéquations. Dans la pile, les inéquations sont séparées par le caractère '!'.

4.4.5. Graphe d'appel

A l'intérieur du module "Constructeur de la pile", l'échange de données entre les procédures et les fonctions se résume par le schéma ci-dessous:



La procédure *ConstrPileIneq* appelle:

♥ *ConstrTabVar*,

- la procédure renvoie la table des variables *tab_var*;

♥ *ConstrTabFonc*,

- la procédure construit la table des fonctions et des opérateurs *tab_fonc*;

♥ *PileIneq*,

- la procédure reçoit comme paramètre de la procédure *ConstrPileIneq* la variable *x* contenant la représentation interne en forme d'un arbre d'une inéquation et *n*, le numéro de début de remplissage dans la pile;
- renvoie la partie de la pile correspondant à une inéquation.

La procédure *Pile Ineq* appelle:

♥ *ConsultTabVar*,

- la procédure reçoit comme paramètre la variable *x* qui est la représentation interne d'une variable simple ou indicée,
- renvoie *i*, la position d'une variable dans la table des variables, ou un message d'erreur signalant l'absence d'une variable dans la table des variables.

♥ *ConsultTabFonc*,

- la procédure reçoit comme paramètre la variable *x* qui est la représentation interne sous forme d'arbre d'une fonction ou d'un opérateur,

- renvoie *i*, la position d'une fonction ou d'un opérateur dans la table des fonctions, ou un message d'erreur signalant l'absence d'une fonction dans la table des fonctions.

Chapitre 5. IMPLEMENTATION DES ALGORITHMES

5.1. Conception

Le rôle du module "Algorithme" est double. D'une part, il constitue l'un des modules de l'interpréteur; d'autre part, il fournit deux méthodes de calcul.

On peut diviser le module " Algorithme " en trois parties fonctionnelles:

- la partie " Calcul ";
- les procédures et fonctions propres à l'algorithme Bottom-Up;
- les procédures et fonctions propres à l'algorithme Top-Down.

Pour les détails d'implémentation des procédures et les fonctions du module " Algorithme ", le lecteur est invité à se reporter à l'annexe 3.

5.2. Calcul d'inéquations

5.2.1. Données partagées par les algorithmes

Les algorithmes Bottom-Up et Top-Down font tous deux appel à la procédure calcul, qui effectue le calcul des inéquations.

Structure des données partagées par les algorithmes:

```
type
   ty_borne=record
                  debut :   integer;
                  fin   :   integer
            end;

   chain_var=^el_chain;
   el_chain = record
                  num_eq    : integer;
                  borne_eq  : ty_borne;
                  chsuiv            :
chain_var;
              end;
```

```
    ty_chain=array[1..    NbVar]    of
chain_var;

    el_reg=record
                val_reg   : integer;
                libre     : boolean;
          end;
    ty_reg=array[1..LongReg] of el_reg;
```

La procédure ParcourPile prépare l'implémentation d'une procédure de calcul. Pour ce faire, elle construit la chaîne représentant une fourchette de calcul, cette chaîne permettant de trouver la sous-pile correspondant à une inéquation.

Nom: ♦ *ParcourPile*;

Paramètres: ◊ *pile* de type ty_chaine;

Effet: • construit un tableau de chaînes de dépendance statique *chain_gen*
{ la variable du membre gauche de l'inéquation correspondant à la tête d'une chaîne et toutes les variables dont elle dépend constituent le reste de la chaîne };
• construit un tableau *par_g* de correspondance entre le tableau des variables tab_var et la position d'une variable dans le système d'inéquations.

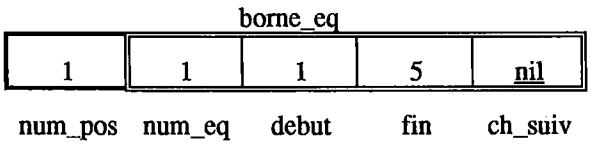
Exemple:

Etant donnés

- un programme SYSTINEQ (la page 46),
- la représentation interne sous forme de pile comme donnée à la page 56,

alors

- *chain_gen[1]* a la forme:



Cette représentation possède l'information de calcul suivante :

◊ le numéro d'inéquation à calculer est égal à 1,

◊ le début de calcul dans la pile est égal à 1,

◊ la fin de calcul dans la pile est égale à 9.

5.2.2. Procédure calcul

La procédure de calcul travaille avec une partie de la pile indiquée par la fourchette de calcul (variable *borne*, passée comme paramètre).

Nom: ♦ *calcul*;

Paramètres: ◊ *pile* de type ty_tab_in,

◊ *borne* de type ty_borne

Effet: • renvoie le résultat de calculs *t*;

• *compteur* {indique le nombre de calculs effectués}

Exemple:

Etant donnée l'équation x[1]=max(5, x[5]+1),

le déroulement d'une procédure calcul peut être présenté par le schéma suivant :

Présentation:

{La procédure utilise une structure auxiliaire reg de type ty_reg}

◊ Était initial du registre:

	true		true		true		true		true
val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre

Pas 1.

◊ Dépiler:

const	integer	5
iconstr	constype	consval

◊ Charger la valeur de la constante dans le registre

5	false		true		true		true		true
val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre

Pas 2.

◊ Dépiler:

indvar	5	integer	5
iconstr	num_var	var_type	ind

◊ Charger la valeur de la variable x[5][7] dans le registre :

5	false	0	false		true		true		true
val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre

Pas 3.

◊ Dépiler:

const	integer	1
iconstr	constype	consval

◊ Charger la valeur de la constante dans le registre

5	false	0	false	1	false		true		true
val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre

Pas 4.

◊ Dépiler:

binop	4	integer	2
iconstr	num_op	op_type	nbr_arg

◊ Calculer l'expression x[5]+1:

- les arguments de la fonction se trouvent dans les cellules 2 et 3 du registre;
- l'opérateur "+" a le numéro 4 dans la table des fonctions;
- les calculs une fois effectués, le registre devient:

5	false	1	false		true		true		true
val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre

[7] On initialise toutes les variables de l'équation à la valeur bottom (0).

Pas 5.

◊ Dépiler:

fonction	15	integer	2
iconstr	num_op	op_type	nbr_arg

◊ Calculer la fonction max(5, x[5]+1):

- les arguments de la fonction se trouvent dans les cellules 1 et 2 du registre;
- la fonction max correspond au numéro 15 dans la table des fonctions;
- après calcul, le registre devient:

5	false		true		true		true		true
val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre	val_reg	libre

◊ Le résultat du calcul pour l'équation est égal à 5.

5.3. Algorithme Bottom-Up

5.3.1. Structure de données

Aux structures de données qu'il partage avec l'algorithme Bottom-Up, l'algorithme Top-Down ajoute la structure auxiliaire de *stack* :

```
type
 ty_stack=^el;
 el= record
     numvar  : integer;
     varsuiv : ty_stack;
 end;
```

5.3.2. Procédure AlgBotUP

L'idée d'un algorithme ascendant Bottom-Up est réalisée par la procédure AlgBotUp.

Procédure AlgBotUp

Nom: ♦ *AlgBotUp*;

Paramètres: ◊ *pile* de type ty_tab_in;

◊ *chain_gen* de type ty_chain;

◊ *chain* de type ty_chain;

Effet: • renvoie la table value comme résultat du calcul des points fixes d'un système d'inéquations;

L'algorithmeBottom-Up se divise en deux étapes:

- étape d'initialisation,
- étape de clôture.

Étape d'initialisation

L'étape d'initialisation effectue les premiers calculs d'inéquations. A ce stade, la procédure *AlgBotUp*

◊ utilise: le tableau chain_gen;

◊ calcule: dans l'ordre d'apparition des inéquations;

◊ crée: le stack des variables dont la valeur a changé par rapport à la valeur intiale (bottom).

Étape de clôture

Elle effectue le calcul des inéquations dont le membre gauche correspond à des variables de *stack.*

A cette étape la procédure

- utilise: le tableau *chain* construit par la procédure *ConstrChain*;
- calcule: selon l'ordre des variables dans *stack*;
- modifie: *stack* (dépile le sommet de *stack*, empile une variable changée au cours des exécutions).

Nom: ♦ *ConstrChain*;

Paramètres: ◊ *pile* de type ty_tab_in;

◊ *NbPar* de type integer;

Effet: • renvoie le tableau *chain* de type ty_chain

{ Chaque élément de la chaîne contient la suite de fourchettes qui indiquent la position des inéquations d'intérêt dans la pile. L'intérêt est de trouver toutes les inéquations, telle que la variable avec le numéro égale à un numéro dans le tableau de chaînes ,figure comme opérande dans leurs parties droites }

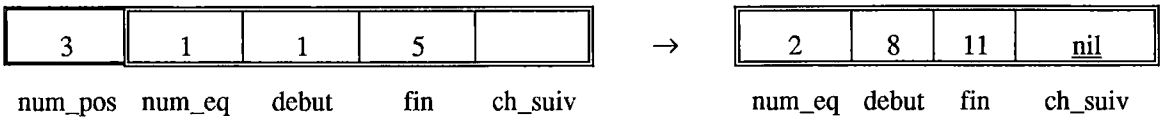
Exemple:

Etant donnés :

- un système d'équations (voir page 45),

- la représentation interne sous forme de pile comme donnée à la page 56.

A la variable x[3] correspond une chaîne *chain[3]*:



5.4. Algorithme Top-Down

5.4.1. Structure de données

La structure de données spécifique à l'algorithme Top-Down est la suivante:

```
type
el_pt=record
      valeur       : integer;
      num_var      : integer;
end;
ty_pt=array[1..NbVar] of el_pt;

el_dg=^suite_dg;
suite_dg=record
         num_var     :integer;
         el_suiv     :el_dg
end;
ty_dg=array[1..NbVar] of el_dg;

ty_optimal=array[1..NbVar] of integer;

ty_ics=array[1..NbVar] of boolean;
```

5.4.2. Dépendance statique

L'idée de dépendance statique, de construction d'une structure qui n'est pas changée au cours des exécutions, est réalisée par la procédure *ConstrSuit*.

Nom: ♦ *ConstrSuit*;

Paramètres: ◊ *pile* de type ty_tab_in;

◊ *chain_gen* de type ty_chain;

Effet: • renvoie *suite* de type ty_dg, la table des chaînes de dépendance statique
{la position dans le tableau correspond au numéro de la variable dans la table des variables, la suite comprend tous les éléments qui l'influencent de façon directe }

Exemple:

Etant donné l'équation x[1]= max(x[2],x[3]),

alors, la suite de dépendance statique construit par la procédure *ConstrSuit* a la forme:

5.4.3. Dépendance dynamique

La structure de graphe de dépendance est construite par deux procédures:

• la procédure *CreerDg* (ajoute un élément au graphe),

• la procédure *MettreNull* (enlève des éléments du graphe).

Ces deux procédures manipulent les tableaux *dg*, *optimal*. Les tableaux *dg*, *optimal* et *ics* permettent de définir l'état des calculs de façon unique et non ambiguë.

Procédure CreerDg

Nom: ♦ *CreerDg*;

Paramètres: ◊ *y* de type integer;
◊ *x* de type integer;

Effet: • rajoute la variable *x* dans la chaîne numéro *y* du graphe de dépendance;
de façon formelle: $dg = dg_0 \setminus \{dg_0(y)\} \cup \{dg_0(y) \cup \{x\}\}$;

Procédure MettreNul

Nom: ♦ *MettreNul*;

Paramètres: ◊ *x* de type integer;

Effet: • si $dg^+(x)$ est le plus petit sous-ensemble de codomaine de dg, tel que :
$\rightarrow y \in dg(x) \Rightarrow y \in dg^+(x)$
$\rightarrow y \in dg(x)$ & $z \in dg^+(y) \Rightarrow z \in dg^+(x)$
alors $dg = dg_0 \setminus \{S \in dg_0 \mid \{y_1, ..., y_n\} \cap dg_0^+(x) \neq \emptyset\}$;

Nous illustrerons les procédure *CreerDg* et *MettreNul* par une exemple concret (cf. infra).

5.4.4. Procédure eval

Nom: ♦ *eval*;

Paramètres: ◊ *x* de type integer;

Effet: • renvoie la table *pt* avec les valeurs de points fixes calculées pour une variable demandée par l'utilisateur, et un groupe de variables nécessaires pour effectuer le calcul de point fixe pour la première; les autres variables ayant la valeur bottom.

Exemple:

Etant donné le programme SYSTINEQ,

```
program essai;
var
x : array [1..3] of integer;
begin
      x[1]=comp(x[2]);
      x[2]=min(3,x[3]+1);
      x[3]=max(x[1],x[2]);
end.
?
```

on veut calculer le point fixe pour la variable x[1]

Présentation[8] :

A. →Après initialisation :

ics[1]=false,

optimal[1]=0,

→On entre dans le boucle "if"

domcodom vaut

num_var	var_suiv
2	nil

→Assigner à pt[1].value la valeur de bottom(=0).

[8] Pour épargner du temps au lecteur, nous ne présentons pas la démonstration complète d'un exemple de calcul de point fixe par l'algorithme Top-Down. Seules sont données les étapes essentielles du déroulement.

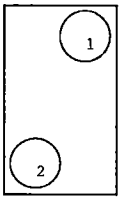
→Entrer dans la boucle "repeat".

→Assigner à optimal [1] la valeur 1.

→La chaîne codom n'est pas vide.

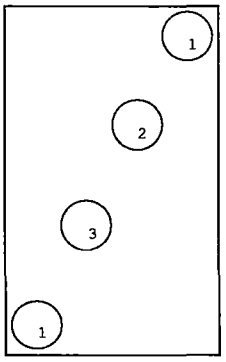
B. →Appeler de façon récursive la procédure *eval* pour la variable numéro 2.

→L'arbre de simulation devient[9] :



De façon semblable à la présentation d'une partie d'une procédure marquée par la lettre **A.** On lance les calculs pour les variables x[3].

→L'arbre de simulation devient:



→ics[1]=true,

optimal[1]=1,

L'appel récursif à la procédure *eval* pour la variable numéro 1 est terminé.

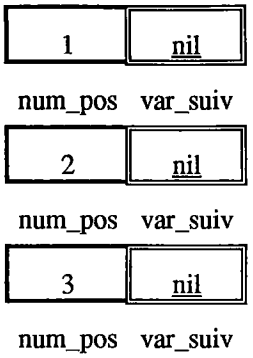
On revient à l'intérieur de l'appel de procédure *eval* pour la variable numéro 3.

→Appeler la procédure *CreerDg* avec les paramètres 1 et 3 qui correspondent aux variables numéros 1 et 3. Au terme de la procédure *CreerDg*, domaine est égale à 1, et 3 correspond au codomaine à ajouter dans le graphe de dépendance.

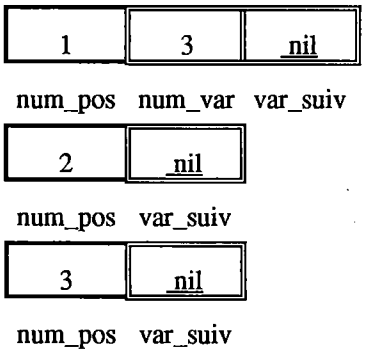
Avant l'appel de la procédure, dg a la forme:

---

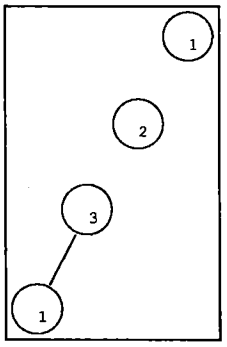
[9] La simulation des variables est représentée graphiquement par un arbre de simulation. Les feuilles de cet arbre sont des cercles où figurent les numéros des variables correspondantes.



Après l'appel de la procédure *CreerDg*, le graphe de dépendance devient:



→L'arbre de simulation prend la forme[10] :



→La chaîne codom n'est pas vide.

Appeler de façon récursive la procédure *eval* pour la variable numéro 2.

→ics[2]=true,

optimal[2]=1,

L'appel récursif à la procédure *eval* pour la variable numéro 2 est terminé.

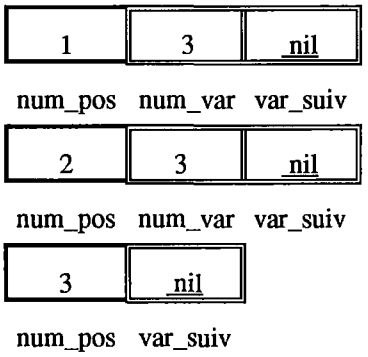
On revient à l'intérieur de l'appel de procédure *eval* pour la variable numéro 3.

---

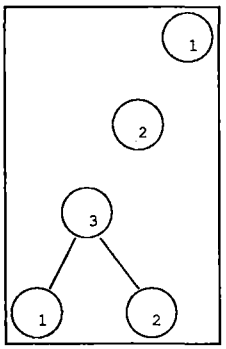
[10] La barre entre deux feuilles d'un arbre de simulation est utilisée pour montrer qu'un élément du codomaine a été ajouté par la procédure *CreerDg*.

→Appeler la procédure *CreerDg* avec les paramètres 2 et 3 qui correspondent aux variables numéros 2 et 3.

Après appel de la procédure *CreeDg*, le graphe de dépendance devient:



→L'arbre de simulation devient:



→Assigner à la variable *num* le numéro d'équation correspondant à la variable numéro 3, *num* est égal à 3;

→Trouver la fourchette de calcul dans la pile.

Assigner la fourchette à la variable borne.

→Appeler la procédure *calcul*, passer comme paramètres *pile* et *borne*,

le résultat des calculs est égal à 0;

→Le résultat des calculs n'est pas différent de pt[3].value,

l'appel récursif de la procédure *eval* pour la variable numéro 3 est terminé,

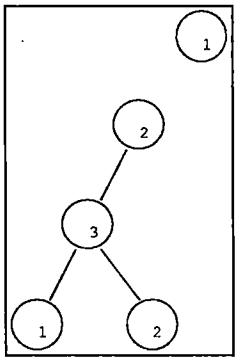
on sort de la proédure avec false comme valeur d'ics[3];

→Appeler la procédure *CreerDg* avec les paramètres 3 et 2 qui correspondent aux variables numéros 3 et 2;

après appel de la procédure *CreeDg*, le graphe de dépendance devient:

1	3	nil
num_pos	num_var	var_suiv
2	3	nil
num_pos	num_var	var_suiv
3	2	nil
num_pos	num_var	var_suiv

→L'arbre de simulation devient:



→Assigner à la variable *num* le numéro d'équation correspondant à la variable numéro 2,
num est égal à 2;
→Trouver la fourchette de calcul dans la pile,
assigner la fourchette à la variable borne;
→Appeler la procédure *calcul* , passer comme paramètres pile et borne,
le résultat de_ calculs est égal à 1;
→Le résultat des calculs est supérieur à pt[2].value,
on entre dans la boucle "if";
→Assigner à pt[2].value le résultat du calcul précédent,
→dg[2] n'est pas nil,
appeler la procédure *MettreNul*, passer comme paramètre le numéro de variable 2, qui correspond au domaine du codomaine à annuler:

Itération 1: domaine est égal à 2⇒ on enlève codomaine (3),
Mettre optimal[2] à 0;
Itération 2: domaine est égal à 3⇒ on enlève codomaine (2),
Mettre optimal[2] à 0;
Itération 3: domaine est égal à 2⇒ il n'y a rien à faire (le codomaine a déjà été enlevé).

Après appel de la procédure *MettreNul*, le graphe de dépendance devient:

1	3	nil

num_pos num_var var_suiv

2	nil

num_pos var_suiv

3	nil

num_pos var_suiv

Le tableau optimal contient les valeurs suivantes:

optimal[1]=1,

optimal[2]=0,

optimal[3]=0,

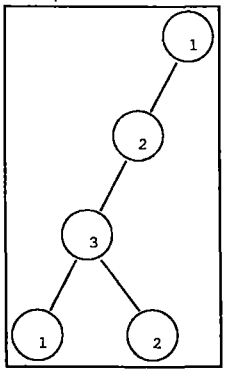
→La valeur pt[2].value a changé au cours des calculs, elle n'est pas optimale.

On relance la simulation pour la variable numéro 2 de façon similaire au processus **B**, à la différence que :

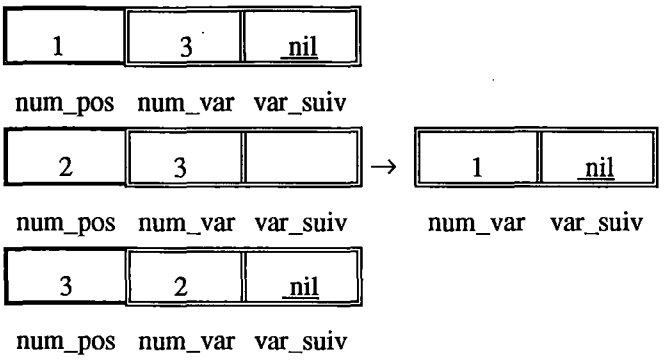
• Une partie du graphe de dépendance est déjà construit; la procédure *CreerDg* contrôle si un élément appartient déjà ou non au codomaine.

• Le résultat du calcul augmente chaque fois d'une unité, jusqu'à atteindre la valeur 3 pour les variables numéros 2 et 3.

C. →L'arbre de simulation a la forme:



Ce qui équivaut à la structure suivante du graphe de dépendance :



→Assigner à la variable *num* le numéro d'équation correspondant à la variable numéro 1, *num* est égal à 1;

→Trouver la fourchette de calcul dans la pile,

assigner la fourchette à la variable borne;

→Appeler la procédure *calcul*, passer comme paramètres pile et borne,

le résultat des calculs est égal à 3;

→Le résultat de calcul est supérieur à pt[1].value,

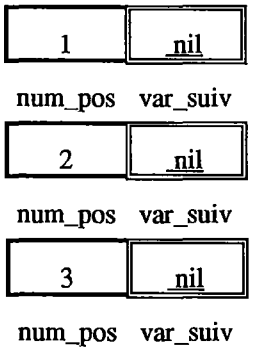
on entre dans la boucle "if";

→Assigner à pt[1].value le résultat du calcul précédent,

→dg[1] n'est pas nil,

appeler la procédure *MettreNul*, passer comme paramètre le numéro de variable 1,

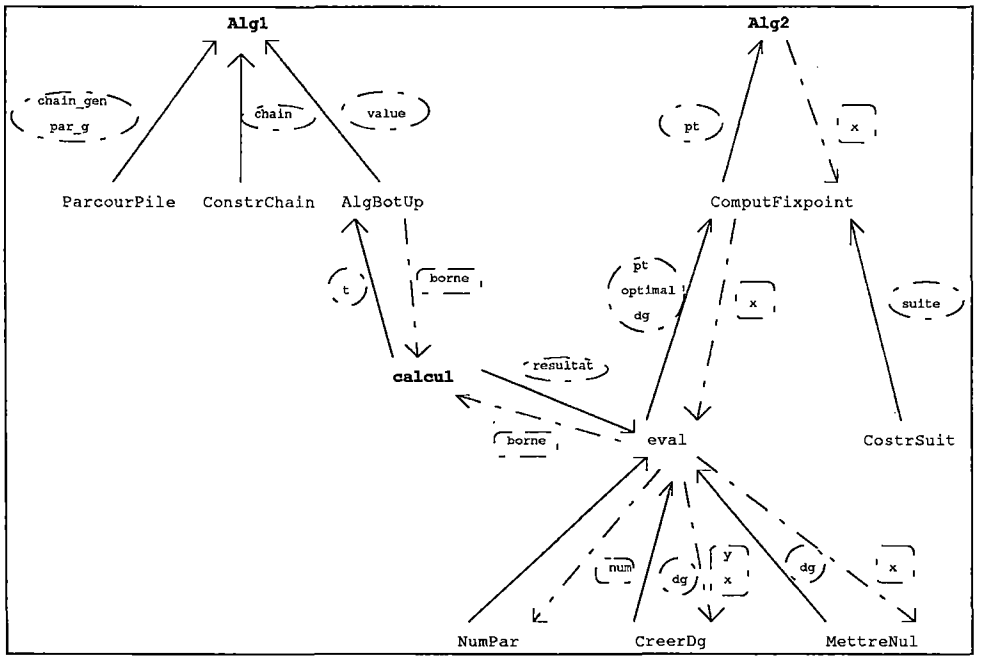
après l'appel de la procédure *MettreNul*, le graphe de dépendance devient:



Tous les éléments de la table optimal sont mis à 0.

D. On relance l'appel de procédure *eval* pour la variable numéro 1. Le déroulement est celui décrit au point **A**, à la différence que la simulation est lancée seulement une fois pour chaque variable (toutes les valeurs ont atteint la valeur de point fixe).

5.5. Graphe d'appel



La procédure *Alg1* appelle:

♥ *ParcourPile*, la procédure

•renvoie à *Alg1*

le tableau *chain_gen* de type ty_chain,

le tableau *par_g* de type ty_int;

♥ *ConstrChain*, la procédure

•renvoie le tableau *chain* de type ty_chain;

♥ *AlgBotUp*, la procédure

•renvoie le tableau *value* de type ty_int.

La procédure *AlgBotUp* appelle:

♥ *calcul*, la procédure

•reçoit la variable *borne* de type ty_borne,

•renvoie la variable *t* de type integer;

La procédure *Alg2* appelle:

♥ *ConstrFixpoint*, la procédure

•reçoit la variable *x* de type integer

•renvoie le tableau *pt* de type ty_pt;

La procédure *Fixpoint* appelle:

♥ *ConstrSuit*, la procédure

- renvoie le tableau *suite* de type ty_dg;

♥ *eval*, la procédure

- reçoit la variable *x* de type integer,
- renvoie le tableau *pt* de type ty_pt,
  dg de type ty_dg,
  optimal de type ty_optimal;

La procédure *eval* appelle:

♥ *calcul*, la procédure

- reçoit la variable *borne* de type ty_borne,
- renvoie la variable *resultat* de type integer;

♥ *NumPar*, la fonction

- reçoit la variable *num* de type integer,
- est de type integer;

♥ *CreerDg*, la procédure

- reçoit les variables *y* et *x* de type integer,
- renvoie le tableau *dg* de type ty_dg;

♥ *MettreNul*, la procédure

- reçoit la variable *x* de type integer,
- renvoie le tableau *dg* de type ty_dg.

Chapitre 6. ETUDE EXPERIMENTALE

6.1. Présentation générale

Le but de ce chapitre est d'étudier le comportement des algorithmes Bottom-Up et Top-Down sous l'angle de leur efficacité.

Pour l'étude expérimentale, nous avons choisi deux exemples de systèmes d'équations. Le premier exemple d'un système d'équations confirme les prévisions théoriques du fait que dans la plupart des cas, l'algorithme Top-Down se comporte de façon plus efficace. Sa conception permet d'éviter des calculs redondants et donc d'économiser du temps de calcul. Le deuxième exemple d'un système d'équations démontre que l'algorithme Top-Down n'est pas toujours "meilleur" par rapport à l'algorithme Bottom-Up,

car l'efficacité de l'algorithme Top-Down dépend non seulement de la structure du système d'équations, mais également de la variable choisie pour évaluer des calculs de point fixe.

6.2. Exemples

Les mesures de complexité sont prises pour deux exemples de systèmes d'équations. Sur base de ces exemples de systèmes d'équation, on a construit quatre exemples de programmes SYSTINEQ:

Exemple 1:

Soit un système d' inéquations:

$$\begin{cases} x_2 = x_1 \pm 1 \\ \quad . \\ \quad . \\ \quad . \\ x_n = x_{n-1} \pm 1 \\ x_1 = 1 \end{cases}$$

où

◊ $T_1 = \ldots = T_n = \{0, \ldots, n\}$;

◊ la rélation d'ordre est '<':

$0 < \ldots < n$;

◊ $x \pm y = \min(n, x+y)$

Pour n=10 le système d'équations en syntaxe du programme SYSTINEQ peut être réécrit comme suit:

```
program boucle1;
  var
    x:array[1..10] of integer;
   begin
      x[2]=min(10,x[1]+1);
      x[3]=min(10,x[2]+1);
      x[4]=min(10,x[3]+1);
      x[5]=min(10,x[4]+1);
      x[6]=min(10,x[5]+1);
      x[7]=min(10,x[6]+1);
      x[8]=min(10,x[7]+1);
      x[9]=min(10,x[8]+1);
      x[10]=min(10,x[9]+1);
      x[1]=1;
   end.
  ?
```

Exemple 2:

Soit un systéme d'équations :

$x_1 = \max(x_2, x_{n-2})$

.

.

.

$x_{n-3} = \max(x_{n-2}, x_{n-2})$

$x_{n-2} =$ if $x_{n-1} = n$ then n else 0

$x_{n-1} = x_n \pm 1$

$x_n = \max(x_1, x_{n-1})$

Si n=10 le système d'équations sera représenté par le programme SYSTINEQ suivant:

```
program boucle2;
   var
    x:array[1..10] of integer;
   begin
      x[1]=max(x[2],x[8]);
      x[2]=max(x[3],x[8]);
      x[3]=max(x[4],x[8]);
      x[4]=max(x[5],x[8]);
      x[5]=max(x[6],x[8]);
      x[6]=max(x[7],x[8]);
      x[7]=max(x[8],x[8]);
      x[8]=comp(x[9]);
      x[9]=min(10,x[10]+1);
      x[10]=max(x[1],x[9]);
   end.
 ?
```

Exemple 3:

Pour l'exemple 3 on utilise le même système d'équations que pour l'exemple 1, en inversant l'ordre d'apparition d'équations.

Pour n=10 on a le programme SYSTINEQ:

```
program boucle3;
  var
   x:array[1..10] of integer;
  begin
     x[1]=1;
     x[10]=min(10,x[9]+1);
     x[9]=min(10,x[8]+1);
     x[8]=min(10,x[7]+1);
     x[7]=min(10,x[6]+1);
     x[6]=min(10,x[5]+1);
     x[5]=min(10,x[4]+1);
     x[4]=min(10,x[3]+1);
     x[3]=min(10,x[2]+1);
     x[2]=min(10,x[1]+1);
  end.
 ?
```

Exemple 4:

Le quatrième exemple représente le système d'équations de l'exemple 1, seulement l'ordre d'apparition est changé selon les règles suivantes: d'abord, on écrit les inéquations avec la partie gauche égale à x[i], telle que i mod 10=0, ensuite i mod 10=1, et ainsi de suite jusqu'à i mod 10=9. Dans chaque groupe d'équations l'ordre d'équation est croissant.

Si n=10, le système d'équations donne le programme ci-dessous:

```
program boucle4;
   var
    x:array[1..10] of integer;
   begin
      x[10]=min(10,x[9]+1);
      x[1]=1;
      x[2]=min(10,x[1]+1);
      x[3]=min(10,x[2]+1);
      x[4]=min(10,x[3]+1);
      x[5]=min(10,x[4]+1);
      x[6]=min(10,x[5]+1);
      x[7]=min(10,x[6]+1);
      x[8]=min(10,x[7]+1);
      x[9]=min(10,x[8]+1);
   end.
 ?
```

Pour ces quatre exemples, la complexité asymptotique de l'algorithme Bottom- Up et de l'algorithme Top-Down est égale à O(mh), où

- ◊ m - la taille du problème (pour rappel: la taille d'un problème est égale à la somme des tailles des arbres fonctionnels d'équations),
- ◊ h - la hauteur du treillis.

6.3. Tableaux

Le tableau récapitulatif ci-dessous présente le temps d'exécution nécessaire pour chaque algorithme , et le nombre d'opérations de calcul effectuées y correspondant.

nom d'algorithme	n	temps (h:min:sec:cen_sec)	nombre d' opérations
Bottom-Up	25	0:0:0:11	351
Top-Down	25	0:0:0:33	121
Bottom-Up	50	0:0:0:66	726
Top-Down	50	0:0:0:22	246
Bottom-Up	75	0:0:1:4	1101
Top-Down	75	0:0:0:38	371
Bottom-Up	100	0:0:1:32	1476
Top-Down	100	0:0:0:50	496
Bottom-Up	125	0:0:1:64	1851
Top-Down	125	0:0:0:66	621
Bottom-Up	150	0:0:2:4	2226
Top-Down	150	0:0:0:77	746

Tableau 1. Les mesures de performances pour l'exemple 1

nom d'algorithme	n	temps (h:min:sec:cen_sec)	nombre d' opérations
Bottom-Up	25	0:0:0:66	453
Top-Down	25	0:0:0:49	352
Bottom-Up	50	0:0:1:27	928
Top-Down	50	0:0:0:93	702
Bottom-Up	75	0:0:2:9	1403
Top-Down	75	0:0:1:48	1052
Bottom-Up	100	0:0:2:69	1878
Top-Down	100	0:0:2:3	1402
Bottom-Up	125	0:0:3:25	2353
Top-Down	125	0:0:2:58	1752
Bottom-Up	150	0:0:3:90	2828
Top-Down	150	0:0:3:19	2102

Tableau 2. Les mesures de performances pour l'exemple 2

Pour le tableau 1 et tableau 2:

◊ la première colonne contient le nom de l'algorithme à étudier,

◊ la deuxième colonne indique la valeur de n choisie,

◊ la colonne 'temps' indique le temps d' exécution nécessaire pour chacun des algorithmes,

◊ la dernière colonne contient le compteur du nombre d'opérations réalisées par chaque l'algorithme.

numéro de variable	nombre d' opérations
1	1
20	96
50	246

Tableau 3. Rapport de dépendance entre le nombre de calculs nécessaires et le choix du numéro de la variable à évaluer(exemple1, algorithme Top-Down)

numéro de variable	nombre d' opérations
1	702
10	739
30	799
48	702
49	7701
50	7701

Tableau 4. Rapport dedépendance entre le nombre de calculs nécessaires et le choix du numéro de la variable à évaluer(exemple 2, algorithme Top-Down)

Pour les tableau 3 et 4:

◊ la première colonne correspond au numéro de variable pour laquelle on cherche le point fixe,

◊ la seconde colonne accumule le nombre d'opérations effectuées par l'algorithme Top-Down.

nom d'algorithme	n	temps (h:min:sec:cen_sec)	nombre d' opérations
Bottom-Up	50	0:0:0:44	491
Top-Down	50	0:0:0:28	246
Bottom-Up	100	0:0:2:99	991
Top-Down	100	0:0:0:49	496

Tableau5. Les mesures de performances pour l'exemple 3

nom d'algorithme	n	temps (h:min:sec:cen_sec)	nombre d' opérations
Bottom-Up	50	0:0:0:99	971
Top-Down	50	0:0:0:28	246
Bottom-Up	100	0:0:2:85	3196
Top-Down	100	0:0:0:49	496

Tableau6. Les mesures de performances pour l'exemple 4

Pour les tableaux 5 et 6:

◊ la première colonne contient le nom de l'algorithme à étudier,

◊ la deuxième colonne indique la valeur de n choisie,

◊ la colonne 'temps' indique le temps d'exécution nécessaire pour chacun des algorithmes,

◊ la dernière colonne contient le compteur du nombre d'opérations réalisées par chaque l'algorithme.

6.4. Analyse des résultats

6.4.1. Exemple 1

Les résultats de tableau 1 démontrent l'avantage de l'algorithme Top-Down par rapport à l'algorithme Bottom-Up grâce au principe de son organisation, c'est-à-dire, d'éviter de relancer les calculs pour les variables qui ont la valeur optimale. L'étude expérimentale démontre que ce principe économise le temps de machine. La transformation représentée par le système d'équations

de cet exemple a la propriété d'être 'sparse'.Nous avons dit dans le chapitre 2 que cette propriété permettra à l'algorithme Top-Down de devenir très efficace, de faire juste le nombre de calculs nécessaires pour arriver à trouver le point fixe.
Le tableau 3 illustre la forte dépendance entre le nombre de calculs effectués par l'algorithme Top-Down et le numéro de variable pour laquelle on calcule le point fixe. La dépendance entre l'indice d'une variable choisie et le nombre de calculs est compréhensible: plus l'indice d'une variable est grand, plus le groupe des variables "participants" dans le calcul devient important parce que la simulation doit être lancée pour toutes les variables d'indice plus petites ou égales à celle pour laquelle on cherche le point fixe.

6.4.2. Exemple 2

Remarquons que la transformation τ pour le deuxième exemple n'est pas du tout 'sparse': pour calculer le point fixe de n'importe quelle variable , on doit lancer la simulation pour toutes les variables présentes dans le système d'équations.
Malgré le faible degré de 'sparsness' de l'algorithme Top- Down, d'après l'examen du tableau 2 on peut avoir l'impression que l'algorithme Top-Down donne un meilleur résultat par rapport à l'algorithme Bottom-Up en ce qui concerne le temps d'exécution et le nombre des opérations effectuées. Cette proposition reste vraie pour toutes les variables sauf les deux dernières. Pour n=10 ce sont les variables numéro 9 et numéro 10, pour n=50 les variables 'exceptions' sont les variables x[49]et x[50] et ainsi de suite. Pour les variables citées, on remarque la 'chute' de temps de calcul.
De façon intuitive, on peut expliquer ce phénomène par le fait, que si on lance le calcul pour n'importe quelle variable à l'exception des deux dernières, l'arbre de simulations déjà en route sera représenté par un arbre avec deux feuilles en bas correspondant aux deux dernières variables, supposons, x[49] et x[50]. Ces deux feuilles x[49]et x[50] ont la propriété d'avoir la dépendance partiellement localisé dans le sens que l'algorithme va boucler autour de ces deux variables jusqu'au moment où elles atteignent la valeur stable de point fixe, et en remontant dans l'arbre de simulations une après l'autre les variables restantes vont être calculées. Si on lance le calcule pour les variables x[49] ou x[50], alors chaque fois que la valeur d'une variable augmente d'une unité on doit parcourir tout l'arbre de simulation, le temps de calcul devient important. Le tableau 4 montre cette 'chute' du nombre de calculs effectués par l'algorithme Top-Down.

6.4.3. Exemples 3 et 4

L'étude des exemples 3 et 4 (les résultats des calculs sont présentées dans les tableaux 5 et 6) nous permet de tirer la conclusion que la complexité asymptotique de l'algorithme Bottom-Up dépend fortement de la manière dont le 'stack' est rempli. L'ordre d'empilage de 'stack' est lié directement à l'ordre d'apparition des équations dans le système d'équations.

Pour l'exemple 3 du programme SYSTINEQ on a inversé l'ordre d'équations de départ; le résultat en est que l'algorithme Bottom-Up est devenu plus efficace. A titre de comparaison, si n=50, l'exemple 1 donne le résultat de 726 opérations effectuées par l'algorithme, quant à l'exemple 3 , le nombre des opérations effectués est égal à 491.
L'exemple 4 donne le plus 'mauvais' résultat parmi ces trois exemples, qui s'inspirent tous du même système d'équations; seulement l'ordre d'apparition d'équations est modifié.
Par contre, les mêmes exemples démontrent que l'ordre d'apparition des inéquations dans le programme SYSTINEQ n'influence pas l'efficacité de l'algorithme Top-Down.

6.5. Synthèse des résultats

On peut tirer les conclusions suivantes quant à l'efficacité des algorithmes de calcul de point fixe Top-Down et Bottom-Up pour une transformation sous forme de système d'équations :

- l'algorithme Top-Down est plus efficace que l'algorithme Bottom-Up pour les systèmes d'inéquations avec un degré de 'sparsness' élevé;
- la complexité asymptotique de l'algorithme Top-Down est influencée par le choix de la variable à développer pour calculer le point fixe;
- l'efficacité de l'algorithme Bottom-Up est conditionnée par l'ordre d'apparition d'équations dans le système d'équations à étudier.

Chapitre 7. MODE D'EMPLOI

7.1. Unités

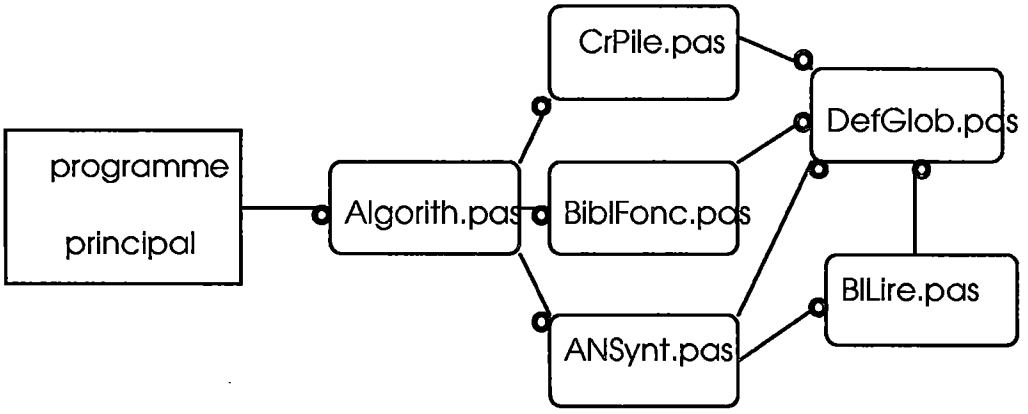
Pour faciliter la tâche de l'utilisateur, l'interpréteur et les deux algorithmes implémentés à la base de l'interpréteur SYSTINEQ sont organisés en 5 unités et en un programme principal. La division des structures de données des procédures et fonctions est dictée par le désir de regrouper les objets en fonction de leur conception.

DefGlob.pas	La structure des données partagées par toutes les unités et le programme principal
BlLire.pas	Les procédures et fonctions de l'analyseur lexical
AnSynt.pas	Les procédures et fonctions de l'analyseur syntaxique
BiblFonc.pas	La bibliothèque de fonctions disponibles à utiliser par les algorithmes de calcul de points fixes[11]
CrPile.pas	Les procédures et fonctions du constructeur de la pile des opérateurs et opérandes
Algorith.pas	La structure des données utilisées par les algorithmes, les procédures et fonctions spécifiques
SystIneq.pas	Programme principal

[11] La bibliothèque de fonctions le lecteur peut trouver dans l'annexe 4

7.2. L'architecture logique

La relation "utilise" entre des unités et programme principal on peut illustrer par le diagramme suivant:



7.3. Interface Homme|Machine

L'utilisateur a le choix:

- Soit il définit le système d'inéquations à exécuter dans le fichier de texte tout en respectant la syntaxe de l'interpréter SYSTINEQ (le syntaxe d'un programme correspondant à un système d'inéquations est définie dans le chapitre 3 "Syntaxe d'un programme SYSTINEQ");
- Soit il utilise pour faire les calculs quatre exemples d'inéquations proposées en parallèle avec l'interpréteur (la description de ces exemples se trouve dans le chapitre 6"Etude expérimentale").

Si l'utilisateur a choisi de développer un de quatre exemples proposés par l'auteur, le nom de fichier-source doit être défini en fonction d'un exemple choisi:

- pour le premièr → "boucle1";
- pour le deuxième → "boucle2";
- pour le troisième → "boucle3";
- pour le quatrième → "boucle4";

Ensuite

- ♦ il faut introduire le nombre des inéquations n.

Sinon,

- ♦ le nom du fichier source, est choisi arbitrairement par l'utilisateur.

En plus du fichier-source, l'interpréteur propose à l'utilisateur de définir

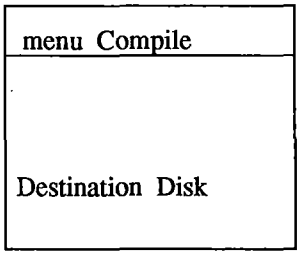
♦ le nom du fichier de destination .

Le fichier de destination contiendra

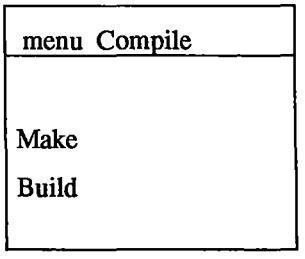
- ♥ la pile des inéquations, la représentation finale des inéquations sous forme de pile,
- ♥ les résultats des calculs: les valeurs de points fixes calculées par deux algorithmes,
- ♥ le temps d'exécution nécessaires et
- ♥ la valeur du compteur qui enregistre le nombre des opérations effectuées par les algorithmes.

7.4. Compilation(Turbo Pascal)

- Charger le fichier SystIneq.pas;
- Choisir l'option de création d'un fichier exécutable;



- Construire le fichier exécutable



7.5. Exécution (DOS)

- Lancer le programme SystIneq.exe;
- Définir le nom du fichier-source;
- Définir le nom du fichier-déstination.

CONCLUSION

Dans le cadre de ce mémoire, nous avons implémenté deux algorithmes de calcul de point fixe pour une transformation décrite par un système d'inéquations. Pour effectuer celle-ci, nous avons construit un interpréteur du langage SYSTINEQ.
Dans le chapitre 6, nous avons fait l'étude expérimentale de deux algorithmes implémentés, afin de comparer les résultats des algorithmes appliqués à des exemples spécialement conçus. Ensuite, nous avons identifié les points forts et les désavantages de ces algorithmes.
Dans la suite de cette conclusion nous parlerons de quelques particularités de l'interpreteur du langage SYSTINEQ, ensuite nous reprendrons les remarques à propos de chacun de ces algorithmes et enfin nous ferons la synthèse des résultats globaux de l'étude expérimentale des algorithmes.

1. Interpréteur du langage SYSTINEQ

L'interpréteur du langage SYSTINEQ sert d'utile intermédiaire entre le langage SYSTINEQ et les algorithmes de calcul de point fixe.
Son orientation aux algorithmes concrets de calcul de point fixe définit la représentation interne finale d'un programme SYSTINEQ sous forme de pile d'opérateurs et opérandes, en passant par une représentation interne sous forme d'un arbre des constructions.
Le syntaxe du langage SYSTINEQ, quant à lui, est orienté pour représenter les systèmes d'inéquations uniquement.

2. Algorithmes

2.1. L'algorithme Bottom-Up

L'algorithme Bottom-Up utilise la méthode ascendante de calcul de point fixe et s'inspire de l'algorithme LINCLOSURE. Il est conçu pour calculer le point fixe d'un système d'inéquations monotones.
Le nombre de calculs effectués par l'algorithme dépend fortement de l'ordre d'apparition des inéquations décrites par un programme SYSTINEQ. Il faut souligner que chaque inéquation est calculée par l'algorithme Bottom-Up au minimum 2 fois, à l'exception de cas où après la phase d'initialisation la valeur est égale à bottom.

2.2. L'algorithme Top-Down

L'algorithme Top-Down est de par sa nature un algorithme descendant. Il est local, ce qui signifie qu' il calcule le point fixe pour une des variables présentes dans un système d'équations, représenté sous forme d'un programme SYSTINEQ et répondant aux règles de syntaxe du langage SYSTINEQ. L'algorithme s'inspire d'un algorithme général pour interprétation abstraite. Il a hérité des propriétés de l'algorithme général d'être quasi-direct (les calculs sont localisés par le groupe des variables strictement nécessaires pour trouver le point fixe de la variable en question), l'algorithme utilise une structure de dépendance dynamique, réalisée techniquement sous forme de graphe de dépendance.
L'utilisation des dépendances dynamiques au cours du calcul permet à l'algorithme d'être très efficace pour un grand groupe d'exemples de systèmes d'équations, de pouvoir propager les résultats de simulation sans faire des calculs redondants. L'algorithme Top-Down se caractérise par le fait que le nombre de calculs effectués par l'algorithme est influencé par le choix d'une variable à développer. Ainsi, pour un même système d'équations le temps de calcul pour des variables distinctes peut être fort variable. A l'encontre de l'algorithme Bottom-Up, l'ordre d'apparition des équations dans un programme SYSTINEQ ne produit aucun effet sur l'efficacité de l'algorithme.

3. Synthèse

Des deux algorithmes présentés, l'algorithme Top-Down est beaucoup plus efficace dans la plupart des cas. Les résultats de l'étude expérimentale des algorithmes implementés dans ce travail confirme les prévisions théoriques[12] de complexité et d'efficacité des algorithmes de calcul de point fixe.

REFERENCES BIBLIOGRAPHIQUES

1. Berry G.: Bottom-Up computation of recursive programs. In RAIRO-rouge 10(3), pages 47-82. Dunod, March 1976.

2. Birkhoff, G.: Colloquium Publications. Volume XXV: Lattice Theory. AMS, 1967.

3. Cleaveland R.: Tableau-Based Model Checking in the Propositional Mu-Calculus, Acta Informatica, 1990.

4. Cousot P., and Cousot R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77), pages 238-252, Los Ageles, California, January 1977.

5. Jorgensen N.: Chaotic fixpoint iteration guided by dynamic dependency, in Workshop on Static Analysis (WSA'93), LNCS.

6. Larsen K.G.: Proof systems for Heenessy-Milner logic with recursion, CAAP'88.

7. Larsen K.G.: Efficient local correctness checking , CAV'92, Forthcoming.

8. Le Charlier B.: L'analyse statique de programmes par interprétation abstraite. Nouvelles de la Science et des Technologies, vol. 9, numéro 4, 1991, pp. 19-25.

9. Le Charlier B. and Van Hentenryck P.: Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol.16, nr. 1, January 1994, an extended abstract has appeared in Proc. of the IEEE fourth International Conference on Programming Languages (ICCL'92).

10. Le Charlier B., Degimbe O., Michel L. and Van Hentenryck P.: Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In Cousot P. and all, editors, Proc of the Third International Workshop on static Analysis (WSA'93), Lecture Notes in Computer Science, Pavoda, September 1993. Springer-Verlag.

11. Le Charlier B., Musumbu K. and Van Hentenryck P.: A generic abstract interpretation algorithm and its complexity analysis. In K. Furukawa, editor, Proceedings of the Eighth International Conference on Logic Programming (ICLP'91).

12. Le Charlier B. and Van Hentenryck P.: A universal top-down fixpoint algorithm, Technical Report 92-22, Institute of Computer Science, University of Namur, Belgium, April 1992.

13. O'keefe R.: Finite Fixed-point problems. In J-L. Lassez, editor, Proceedings of the Fourth International Conference on Logic Programming (ICCL'87), pages 729- 743, Melbourne, Australia, May 1987. MIT Press.

14. Vergauwen B., Wauman J., Lewi J.: Efficient FixPoint Computation, in Workshop on Static Analysis (WSA'93), LNCS.

ANNEXES

Annexe 1. BNF DES SYMBOLES DE BASE

L'ensemble des symboles de base est defini, ci-dessous, par une grammaire BNF:

< symbole de base>::=<identificateur>|<constante>|<symbole spécial>

<identificateur>::=<lettre miniscule>|<identificateur><lettre miniscule>|<identificateur><chiffre>

<lettre miniscule>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<chiffre>::=0|1|2|3|4|5|6|7|8|9

<constante>::=<entier>|<caractère>|<valeur de verité>

<entier>::=<chiffre>|<entier><chiffre>

<caractère>::=<lettre majuscule>|<chiffre>|<caractère spécial>

<lettre majuscule>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<caractère spécial>::= |+|-|*|/|(|)|=|,|.|"|!|?|>|<|;|:

<symbole spécial>::=+|-|*|(|)|=|>|<|begin|end|]|[|program|var|mod|not|or|integer|boolean|char

<valeur de vérité>::=true|false

Annexe 2. CODE-SOURCE

fichier DefGlob.pas (la structure de données partagées par toutes les unités et par le programme principal)

```
UNIT DefGlob ;

INTERFACE

CONST
{*************************************************************************}
{**********************Constantes pour les messages d'erreurs*************************}
{*************************************************************************}
  PAS_SS_PREM    ='100';  {pas de symb spec comme premier symbole}
  PAS_PROG_SS    ='101';  {pas de PROGRAM comme symb spec}
  PAS_ID_PROG    ='102';  {pas d'identificateur après PROGRAM}
  PAS_SS_ID      ='103';  {pas de symb spec après l'identificateur}
  PAS_PV_ID      ='104';  {pas de ; après l'identificateur}
  PAS_SS_PV      ='105';  {pas de symb spec après ;}
  PAS_VAR_PV     ='106';  {pas de VAR/BEGIN après ;}
  PAS_SS_DV      ='107';  {pas de symb spec après la décl des variables}
  PAS_PV_DV      ='108';  {pas de; après la décl des variables}
  PAS_SSID_PV    ='109';  {pas de symb spec ou d'identificateur après ;}
  PAS_BEG_PV     ='100';  {pas BEGIN après ;}
  PAS_SS_END     ='100'   {pas de symb spec aprés END}
  PAS_P_END      ='112';  {pas de . après END final}
  PAS_Q_P        ='113';  {pas de ? après . }
  PAS_PVPO_ID    ='114';  {pas de ; ou ( après l'identificateur}
  PAS_SS_TPF     ='115';  {pas de symb spec après type des param form}
  PAS_PF_TPF     ='116';  {pas de ) après type des param form}
  PAS_SS_PF      ='117';  {pas de symb spec après ) }
  PAS_PV_PF      ='118';  {pas de ; après ) }
  PAS_PO_ID      ='119';  {pas de ( après identificateur}
  PAS_DP_PF      ='120';  {pas de : après ) }
  PAS_SS_DP      ='121';  {pas de symb spec après : }
  PAS_CBI_DP     ='122';  {pas de CHAR/BOOLEAN/INTEGER/ARRAY après : }
  PAS_ID_PO      ='123';  {pas d'identificateur après ( }
  PAS_ID_V       ='124';  {pas d'identificateur après , }
  PAS_DP_ID      ='125';  {pas de : après l'identificateur}
  PAS_ID_VAR     ='126';  {pas d'identificateur après VAR}
  PAS_SS_P       ='128';  {pas de symb spec après . }
  SYMB_ERR       ='129';  {symbole erroné}
  PAS_SYMB       ='130';  {présence de ? }
  END_ATTENDU    = '131' ; { Instruction incorrecte : mot réservé"END" attendu}
  PF_ATTENDU     = '132' ; { Caractère ")" attendu }
  EXPD_ATTENDU = '133' ; { Expression de désignation attendue }
  PO_ATTENDU     = '134' ; { Caractère ")" attendu }
  SS_ERR         = '135' ; { Ce symbole spécial ne peut pas débuter une inéquation }
  EG_ATTENDU     = '136' ; { Caractère "=" attendu }
  DP_ATTENDU     = '137' ; { Caractère ":" attendu }
  CF_ATTENDU     = '138' ; { Caractère "]" attendu }
  INS_ERR        = '139' ; { Instruction incorrecte }
  SYMB_PL_ERR    = '140' ; { Ce symbole ne peut pas débuter une inéquation }
  SYMB_PAS_TR    = '141' ; { Symbole de base introuvable }
```

```
  FIN_FICH_ERR    = '142' ; { Fin de fichier non prévue }
  SYMB_FACT_ERR= '143' ; { Ce symbole ne peut pas débuter un facteur }
  PAS_IN_BIBLIO   = '144' ; { N'est pas dans la bibliothèque de fonctions }
  PAS_SS_ENDF     = '145' ; { Pas de symbole spécial après l'END final }
  PAS_TAB_VAR     = '146' ; { Variable non déclarée dans la table de variables }
  THEN_ATTENDU = '147' ; { Mot réservé THEN attendu }
  PAS_SS_ARRAY    = '148' ; { pas de symb spec après ARRAY }
  PAS_CO_ARRAY    = '149' ; { pas de [ après ARRAY}
  PAS_ENT_CO      = '150' ; { pas d'entier binf après [ }
  PAS_SS_ENT      = '151' ; { pas de symb spec après l'entier binf }
  PAS_P_ENT       = '152' ; { pas de . après l'entier bi }
  PAS_P_PREM      = '153' ; { pas de . après le premier . }
  PAS_ENT_DEP     = '154' ; { pas d'entier bsup après le deuxième . }
  PAS_CF_ENT      = '155' ; { pas de ] après entier bsup }
  PAS_SS_CF       = '156' ; { pas de symb spec après ] }
  PAS_OF_CF       = '157' ; { pas d'OF après ] }
  PAS_SS_OF       = '158' ; { pas de symb spec après OF }
  PAS_CBI_OF      = '159' ; { pas de CHAR/BOOLEAN/INTEGER après OF }
  DIV_0           = '160' ; { division par 0 }


{*****************************************************************************}
{************************Déclaration de type**********************************}
{*****************************************************************************}

TYPE

  chaine=STRING[8];

  tconstr = (fonction, proc, appel, simvar, indvar, unop, binop, cons,
  affect,  debut, vari, val, prog,cond1,cond2,tab) ;

  operateur = (fois, quotient,modulo, reste, plus, moins, egal, different,
  inferieur, superieur, infeq, supeq, et, non, ou, maximum,
  minimum,compar) ;

  tbsymb = (identificateur, caractere, booleen, entier, specsymb,
  nosymb, errsymb) ;

  valtype = caractere..entier ;

  ty_lident  = ^lident ;

  lident  = RECORD
    num_ligne  : INTEGER ;
    pident       : chaine ;
    lsident      : ty_lident ;
  END ;


  ty_constr  = ^constr ;
  ty_lconstr = ^lconstr ;

  bsymb =  RECORD
    CASE ibsymb : tbsymb OF

    identificateur, specsymb  :( cval :chaine );
```

```
    caractere,booleen, entier :(consval :INTEGER);
    nosymb, errsymb          :()
  END;

  lconstr = RECORD
    pconstr  : ty_constr ;
    lsconstr : ty_lconstr
  END ;

  constr = RECORD
    num_ligne : INTEGER ;
    CASE iconstr : tconstr OF
    prog :
    (pnom        :chaine ;
     lvar        :ty_lconstr ;
    corps        :ty_constr) ;

    fonction :
    (fnom        :chaine ;
    leff         :ty_lconstr;
    ftype        :valtype);

    simvar :
    (vnom        :chaine);

    indvar :
    (tnom        :chaine ;
    indice       :INTEGER) ;

    unop :
    (uop         :operateur ;
    oper         :ty_constr) ;

    binop :
    (bop         :operateur ;
    oper1, oper2:ty_constr) ;

    cons :
    (constype    :valtype ;
    consval      :INTEGER) ;


    debut :
    (lineq       :ty_lconstr) ;

    vari, val :
    (vartype     :valtype ;
    lvnom        :ty_lident) ;

    cond1,cond2:
    (expr1,instr1,instr2:ty_constr);
    tab      :
    (tabtype     :valtype;
    ltnom        :ty_lident;
    db,fb        :INTEGER)

  END ;
```

```
  message = ARRAY[1..3] OF CHAR ;

VAR
  N,nl          :INTEGER;
  fich_out      :TEXT;


{*************************************************************************}
{******************Fait l'affichage des erreurs syntaxiques***************}
{*************************************************************************}

procedure affichage(num_ligne:integer;msg:message);

  implementation


PROCEDURE affichage (num_ligne:INTEGER;msg: message) ;

  TYPE errmsgtype = RECORD
       msg      : message ;
       txt      : STRING[80] ;
     END ;

  CONST NbMsg = 62;
    tabmsg : ARRAY [1..NbMsg] OF errmsgtype =
    ((msg:Pas_SS_PREM     ;txt:'Pas de symb spec comme premier symbole'),
    (msg:Pas_PROG_SS      ;txt:'Pas PROGRAM comme symb spec'),
    (msg:Pas_ID_PROG      ;txt:'Pas d'identificateur après PROGRAM'),
    (msg:Pas_SS_ID        ;txt:'Pas de symb spec après l'identificateur'),
    (msg:Pas_PV_ID        ;txt:'Pas de ; après l'identificateur'),
    (msg:Pas_SS_PV        ;txt:'Pas de symb spec après ;'),
    (msg:Pas_VAR_PV       ;txt:'Pas de VAR/BEGIN après ;'),
    (msg:Pas_SS_DV        ;txt:'Pas de symb spec après la decl des variables'),
    (msg:Pas_PV_DV        ;txt:'Pas de ; après la decl des variables'),
    (msg:Pas_SSID_PV      ;txt:'Pas de symb spec ou d'identificateur après ;'),
    (msg:Pas_BEG_PV       ;txt:'Pas de BEGIN après ;'),
    (msg:Pas_SS_END       ;txt:'Pas de symb spec après END'),
    (msg:Pas_P_END        ;txt:'Pas de . après END final'),
    (msg:Pas_Q_P          ;txt:'Pas de ? après . '),
    (msg:Pas_PVPO_ID      ;txt:'Pas de ; ou ( après l'identificateur'),
    (msg:Pas_SS_TPF       ;txt:'Pas de symb spec après type des param form'),
    (msg:Pas_PF_TPF       ;txt:'Pas de ) après type des param form'),
    (msg:Pas_SS_PF        ;txt:'Pas de symb spec après ) '),
    (msg:Pas_PV_PF        ;txt:'Pas de ; après ) '),
    (msg:Pas_PO_ID        ;txt:'Pas de ( après l'identificateur'),
    (msg:Pas_DP_PF        ;txt:'Pas de : après ) '),
    (msg:Pas_SS_DP        ;txt:'Pas de symb spec après : '),
    (msg:Pas_CBI_DP       ;txt:'Pas de CHAR/BOOLEAN/INTEGER après : '),
    (msg:Pas_ID_PO        ;txt:'Pas d'identificateur après ( '),
    (msg:Pas_ID_V         ;txt:'Pas d'identificateur après , '),
    (msg:Pas_DP_ID        ;txt:'Pas de : après l'identificateur'),
    (msg:Pas_ID_VAR       ;txt:'Pas d'identificateur après VAR'),
    (msg:Pas_CBI_DP       ;txt:'Pas de CHAR/BOOLEAN/INTEGER/ARRAY après : '),
    (msg:Pas_SS_P         ;txt:'Pas de symb spec après . '),
    (msg:SYMB_ERR         ;txt:'Symbole erroné'),
    (msg:Pas_SYMB         ;txt:'Présence de ? trop tôt'),
    (msg:END_ATTENDU      ;txt:'Instruction incorrecte : mot réservé "END" attendu '),
```

```
    (msg:PF_ATTENDU      ;txt:'Caractère ")" attendu'),
    (msg:EXPD_ATTENDU    ;txt:'Expression de désignation attendue'),
    (msg:PO_ATTENDU      ;txt:'Caractère ")" attendu'),
    (msg:SS_ERR          ;txt:'Ce symbole spécial ne peut pas débuter une instruction'),
    (msg:EG_ATTENDU      ;txt:'Caractère "=" attendu'),
    (msg:DP_ATTENDU      ;txt:'Caractère ":" attendu'),
    (msg:CF_ATTENDU      ;txt:'Caractère "]" attendu'),
    (msg:INS_ERR         ;txt:'Inéquation incorrecte'),
    (msg:SYMB_PL_ERR     ;txt:'Ce symbole ne peut pas débuter une inéquation'),
    (msg:SYMB_Pas_TR     ;txt:'Symbole de base introuvable'),
    (msg:FIN_FICH_ERR    ;txt:'Fin de fichier non prévue'),
    (msg:SYMB_FACT_ERR;txt:'Ce symbole ne peut pas débuter un facteur'),
    (msg:Pas_IN_BIBLIO   ;txt:'Il n'est pas dans la bibliothèque de fonctions'),
    (msg:Pas_SS_ENDF     ;txt:'On ne trouve pas le symbole spécial après END final'),
    (msg:Pas_TAB_VAR     ;txt:'La variable n'est pas declarée dans la table des variables'),
    (msg:THEN_ATTENDU    ;txt:'Mot reservé THEN attendu'),
    (msg:Pas_SS_ARRAY    ;txt:'Pas de symb spec après ARRAY'),
    (msg:Pas_CO_ARRAY    ;txt:'Pas de [ après ARRAY'),
    (msg:Pas_ENT_CO      ;txt:'Pas d'entier binf après ['),
    (msg:Pas_SS_ENT      ;txt:'Pas de symb spec après l'entier binf'),
    (msg:Pas_P_ENT       ;txt:'Pas de . après l'entier bi'),
    (msg:Pas_SS_P        ;txt:'Pas de symb spec après .'),
    (msg:Pas_P_PREM      ;txt:'Pas de . après le premier .'),
    (msg:Pas_ENT_DEP     ;txt:'Pas d'entier bsup après le deuxième .'),
    (msg:Pas_CF_ENT      ;txt:'Pas de ] après entier bsup'),
    (msg:Pas_SS_CF       ;txt:'Pas de symb spec après ] '),
    (msg:Pas_OF_CF       ;txt:'Pas d'OF après ]'),
    (msg:Pas_SS_OF       ;txt:'Pas de symb spec après OF'),
    (msg:Pas_CBI_OF      ;txt:'Pas de CHAR/BOOLEAN/INTEGER après OF'),
    (msg:DIV_0           ;txt:'Division par 0'));
  VAR i:INTEGER;
BEGIN
  write('Erreur:',msg);
  IF (num_ligne>=1) THEN writeln(' la ligne', num_ligne, ':')
  ELSE writeln;

  i:=1;
  WHILE ((i<=NbMsg) AND (tabmsg[i].msg<>msg)) DO i:=i+1;
  IF (i>NbMsg) THEN
    writeln('Erreur inconnue.')
  ELSE
    writeln('  ',tabmsg[i].txt);
  halt(1)
END;
END.{unit}
```

fichier BILire.pas (les procédures et fonctions de l'analyseur lexical)

```
UNIT BLLire ;
INTERFACE

USES  DefGlob;
{
procedure liresymb;
}
```

```
CONST
  nbcar = 56 ;
  tabcar : ARRAY [ 1 .. nbcar ] OF CHAR =
  (' ',
  'A','B','C','D','E','F','G','H','I','J',
  'K','L','M','N','O','P','Q','R','S','T',
  'U','V','W','X','Y','Z',
  '0','1','2','3','4','5','6','7','8','9',
  '+','-','*','/','(',')','=',',','.',':','"','!','',
  '?','>','<',':',';','[',']');

  chiffre   : SET OF CHAR =
  ['0','1','2','3','4','5','6','7','8','9'];

  lettre    : SET OF CHAR =
  ['A','B','C','D','E','F','G','H','I','J','K','L','M',
  'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];

  separateur : SET OF CHAR =
  [' ','+','-','*','(',')','=','.','?','>','<',';',':','[',']','"',
  ',',chr(9), chr(10), chr(13) ] ;

  sign_ver  : ARRAY [1..2] OF chaine =
  ( 'TRUE', 'FALSE' ) ;

  NbSymbSpec = 28 ;

  symbspec  : ARRAY [1..NbSymbSpec] OF chaine =
  ( '+', '-',  '*', '(', ')', '=', '>', '<',
  ':', ';',  'AND', 'BEGIN', 'DIV', 'END',
  'MOD', 'NOT', 'OR', 'PROGRAM', 'VAR',
  'INTEGER', 'BOOLEAN', 'CHAR','ARRAY', 'OF',
  ',',  '.' ,'[', ']' ) ;

VAR
  fich_en       : TEXT ;
  ligne         : INTEGER ;
  symb          : bsymb ;
  caract_pres   : BOOLEAN ;
  caract_mem    : CHAR ;


  implementation

VAR   l_chaine: chaine;
  s             : CHAR;
  fact          : INTEGER ; { Conversion de chaine en entier }
  j             : INTEGER ;

function getchar:char                       ;forward;
procedure putchar(s:char)                   ;forward;
function InSymbSpec (s : chaine) : boolean  ;forward;
function NumTabCar (c : char) : integer     ;forward;
```

```
{*************************************************************************}
{************************** Function getchar*****************************}
{*************************************************************************}

FUNCTION getchar   :CHAR;
  VAR caract       : CHAR ;

BEGIN {getchar} ;

  IF (caract_pres)
    THEN
  BEGIN
    caract := caract_mem;
    caract_pres := FALSE
  END
  ELSE
    read(fich_en, caract) ;

  getchar :=Upcase( caract) ;
END ; {getchar}



{*************************************************************************}
{************************** Procédure putchar ***************************}
{*************************************************************************}

PROCEDURE putchar (s:CHAR);

BEGIN {putchar}
  caract_pres:=TRUE ;
  caract_mem:=s
END ; {putchar}
{-----------------------------------------------------------------
  Renvoie la valeur vraie si specsymbole se trouve dans le tableau
  de specsymboles; sinon, faux
-----------------------------------------------------------------}
FUNCTION InSymbSpec (s : chaine) : BOOLEAN ;
  VAR i: INTEGER ;

BEGIN{InSymbSpec}
  InSymbSpec := FALSE ;
  FOR i:=1 TO NbSymbSpec DO
  BEGIN
    IF symbspec[i] = s THEN
    BEGIN
      InSymbSpec := TRUE ;
      i := NbSymbSpec
    END
  END ;
END ;{InSymbSpec}
{-----------------------------------------------------------------
  Renvoie le numéro de caractère dans la table des caractères
-----------------------------------------------------------------}
FUNCTION NumTabCar (c : CHAR) : INTEGER ;
  VAR i : INTEGER ;
BEGIN{NumTabCar}
```

```
  NumTabCar := 0 ;
  FOR i:=1 TO nbcar DO
  BEGIN
    IF tabcar[i] = c
      THEN
    BEGIN
      NumTabCar := i ;
      i := nbcar
    END
  END ;
END ;{NumTabCar}
{-------------------------------------------------------------------
  Lit les symboles à partir du fichier de texte d'entrée,
  forme la chaîne de caractères, anlyse la chaîne construite
-------------------------------------------------------------------}
PROCEDURE liresymb;

BEGIN {liresymb}
  s := ' ';
  REPEAT
{-------------------------------------------------------------------
  Passer les espaces et fins de lignes
-------------------------------------------------------------------}
    s := getchar ;
    IF s = chr(13) THEN ligne := ligne + 1 ;
  UNTIL ( ( s <> ' ') AND ( s <> chr(13) ) AND
  ( s <> chr(10) ) AND ( s <> chr(9) ) );

{-------------------------------------------------------------------
 Première phase: former une chaîne contenant le symbole à lire
-------------------------------------------------------------------}
  IF ( s IN separateur )
    THEN
  BEGIN
    l_chaine := s ;
    IF ( s = "" ) THEN {* Si constante caractère*}
    BEGIN
      s := getchar ;
      l_chaine := l_chaine + s ;
      s := getchar ;
      l_chaine := l_chaine + s
    END ;
  END
  ELSE
  BEGIN
    l_chaine := '' ;
    WHILE NOT( s IN separateur ) DO
    BEGIN
      l_chaine := l_chaine + s ;
      s := getchar
    END ;
    putchar(s)
  END ;

 {-------------------------------------------------------------------
   Deuxième phase: analyser la chaîne lue
 -------------------------------------------------------------------}
   IF (l_chaine = sign_ver[1]) OR (l_chaine =sign_ver[2])
```

```
    THEN
  BEGIN
    symb.ibsymb:=booleen;
    IF l_chaine='TRUE'
      THEN symb.consval:=1
    ELSE symb.consval:=0
  END
  ELSE
    IF (InSymbSpec (l_chaine))
      THEN

    BEGIN
      symb.ibsymb := specsymb;
      symb.cval  := l_chaine;
    END
    ELSE
      IF l_chaine[1] = ''{*Si constante caractère*}
        THEN
      BEGIN
        IF ( l_chaine[3] <> '' ) THEN
          symb.ibsymb := errsymb
        ELSE
        BEGIN
          symb.ibsymb  := caractere;
          symb.consval := NumTabCar (l_chaine[2])
        END
      END
      ELSE
        IF l_chaine = '?'
          THEN
          symb.ibsymb:=nosymb
        ELSE
          IF l_chaine[1] IN lettre
            THEN
          BEGIN
            symb.ibsymb:=identificateur;
            symb.cval:=l_chaine ;
          END
          ELSE
            IF l_chaine[1] IN chiffre
              THEN
            BEGIN
              symb.ibsymb:=entier;
              fact := 1 ;
              symb.consval := 0 ;
              FOR j:=length(l_chaine) DOWNTO 1 DO
              BEGIN
{-----------------------------------------------------------------
  Conversion d'une chaîne en valeur numérique
-----------------------------------------------------------------}
                symb.consval := symb.consval +
                fact * (Ord(l_chaine[j])-48);
                fact := fact * 10 ;
              END
            END
            ELSE
              symb.ibsymb:=errsymb
END ; {liresymb}
```

```
END. {unit }
```

fichier AnSynt.pas (les procédures et les fonctions de l'analyseur syntaxique)

```
UNIT AnSynt ;
INTERFACE

USES DefGlob,BlLire ;

function sxprog:ty_constr;

VAR
   ligne          : INTEGER;
   progr, ptr_prg : ty_constr;
   sxerreur       : BOOLEAN;
   avance         : BOOLEAN;
   cour           : BOOLEAN;
   implementation

function sxvar    :ty_constr    ;forward;
function sxineq   :ty_constr    ;forward;
function sxfacteur:ty_constr    ;forward;
function sxterme  :ty_constr    ;forward;
function sxaexpr  :ty_constr    ;forward;
function sxneg    :ty_constr    ;forward;
function sxconj   :ty_constr    ;forward;
function sxexpr   :ty_constr    ;forward;


{-------------------------------------------------------------------------
   Construction de la représentation interne,
   des déclarations de variables,
   à partir de la represéntation externe
-------------------------------------------------------------------------}

FUNCTION sxvar:ty_constr;

   VAR
      point_con   : ty_constr;
      nouveau     : ty_lident;
      precedent   : ty_lident;
      tete        : ty_lident;

{-------------------------------------------------------------------------
   Construction de la représentation interne
   d'une chaine de variables de type tableau,
   à partir de  la représentation externe
-------------------------------------------------------------------------}
   PROCEDURE ch_tabvar;

   BEGIN {*ch_tabvar*}
```

```
      liresymb;
      IF NOT( symb.ibsymb=entier )
        THEN
      BEGIN
        sxerreur:=TRUE;
        IF symb.ibsymb=errsymb
          THEN affichage(nl,SYMB_ERR)
        ELSE
          IF symb.ibsymb=nosymb
            THEN affichage(nl,PAS_SYMB)
          ELSE affichage(nl,PAS_ENT_DEP)
      END
{-------------------------------------------------------------------------
  Pas d'entier après le deuxième '.'
-------------------------------------------------------------------------}
      ELSE
      BEGIN
        point_con^.fb:=symb.consval;
        liresymb;
        IF NOT( symb.ibsymb=specsymb )
          THEN
        BEGIN
          sxerreur:=TRUE;
          IF symb.ibsymb=errsymb
            THEN affichage(nl,SYMB_ERR)
          ELSE
            IF symb.ibsymb=nosymb
              THEN affichage(nl,PAS_SYMB)
            ELSE affichage(nl,PAS_SS_ENT)
        END
{-------------------------------------------------------------------------
  Pas de specsymbole après l'entier borne_sup
-------------------------------------------------------------------------}
        ELSE
          IF NOT( symb.cval=']' )
            THEN
          BEGIN
            sxerreur:=TRUE;
            affichage(nl,PAS_CF_ENT)
          END
{-------------------------------------------------------------------------
  Pas de ] après l'entier borne supérieure
-------------------------------------------------------------------------}
          ELSE
          BEGIN
            liresymb;
            IF NOT( symb.ibsymb=specsymb )
              THEN
            BEGIN
              sxerreur:=TRUE;
              IF symb.ibsymb=errsymb
                THEN affichage(nl,SYMB_ERR)
              ELSE
                IF symb.ibsymb=nosymb
                  THEN affichage(nl,PAS_SYMB)
                ELSE affichage(nl,PAS_SS_CF)
            END
```

```
{-------------------------------------------------------------------
  Pas de specsymbole après ]
-------------------------------------------------------------------}
            ELSE
              IF NOT( symb.cval='OF')
                 THEN
              BEGIN
                 sxerreur:=TRUE;
                 affichage(nl,PAS_OF_CF)
              END
{-------------------------------------------------------------------
  Pas d'OF après ]
-------------------------------------------------------------------}
              ELSE
              BEGIN
                 liresymb;
                IF NOT( symb.ibsymb=specsymb )
                   THEN
                BEGIN
                   sxerreur:=TRUE;
                  IF symb.ibsymb=errsymb
                     THEN affichage(nl,SYMB_ERR)
                   ELSE
                     IF symb.ibsymb=nosymb
                        THEN affichage(nl,PAS_SYMB)
                     ELSE affichage(nl,PAS_SS_OF)
                END
{-------------------------------------------------------------------
  Pas de specsymb après OF
-------------------------------------------------------------------}
                ELSE
                  IF NOT (( symb.cval = 'CHAR'    ) OR
                    ( symb.cval = 'BOOLEAN' ) OR
                    ( symb.cval = 'INTEGER' ))
                     THEN
                   BEGIN
                     sxerreur:=TRUE;
                     affichage(nl,PAS_CBI_OF)
{-------------------------------------------------------------------
  Problème dans la déclaration de tableau
-------------------------------------------------------------------}
                   END
                   ELSE
                   BEGIN
                     IF symb.cval='CHAR'
                        THEN
                        point_con^.tabtype:=caractere;
                     IF symb.cval='BOOLEAN'
                        THEN
                        point_con^.tabtype:=booleen;
                     IF symb.cval='INTEGER'
                        THEN
                        point_con^.tabtype:=entier;
                     point_con^.ltnom:=tete;
                     liresymb;
                     sxvar:=point_con
                   END
               END
```

```
          END
      END
    END; {*ch_tabvar*}

{-----------------------------------------------------------------------
  Construction de la représentation interne
  des déclarations de variables,
  à partir de la représentation externe des variables
  de type tableau
-----------------------------------------------------------------------}
  PROCEDURE tab_var;

  BEGIN{*tab_var*}
    point_con^.iconstr:=tab;
    liresymb;
    IF NOT ( symb.ibsymb=specsymb )
      THEN
    BEGIN
      sxerreur:=TRUE;
      IF symb.ibsymb=errsymb
        THEN affichage(nl,SYMB_ERR)
      ELSE
        IF symb.ibsymb=nosymb
          THEN affichage(nl,PAS_SYMB)
        ELSE affichage(nl,PAS_SS_ARRAY)
    END
{-----------------------------------------------------------------------
  Pas de specsymbole après ARRAY
-----------------------------------------------------------------------}
    ELSE
      IF NOT( symb.cval='[' )
        THEN
      BEGIN
        sxerreur:=TRUE;
        affichage(nl,PAS_CO_ARRAY)
      END
{-----------------------------------------------------------------------
  Pas de [ après ARRAY
-----------------------------------------------------------------------}
      ELSE
      BEGIN
        liresymb;
        IF NOT( symb.ibsymb=entier )
          THEN
        BEGIN
          sxerreur:=TRUE;
          IF symb.ibsymb=errsymb
            THEN affichage(nl,SYMB_ERR)
          ELSE
            IF symb.ibsymb=nosymb
              THEN affichage(nl,PAS_SYMB)
            ELSE affichage(nl,PAS_ENT_CO)
        END
{-----------------------------------------------------------------------
  Pas d'entier après [
-----------------------------------------------------------------------}
        ELSE
        BEGIN
```

```
            point_con^.db:=symb.consval;
            liresymb;
            IF NOT( symb.ibsymb=specsymb )
               THEN
            BEGIN
               sxerreur:=TRUE;
               IF symb.ibsymb=errsymb
                  THEN affichage(nl,SYMB_ERR)
               ELSE
                  IF symb.ibsymb=nosymb
                     THEN affichage(nl,PAS_SYMB)
                  ELSE affichage(nl,PAS_SS_ENT)
            END
{-------------------------------------------------------------------
  Pas de specsymb après entier borne inférieure
-------------------------------------------------------------------}
            ELSE
               IF NOT( symb.cval='.')
                  THEN
               BEGIN
                  sxerreur:=TRUE;
                  affichage(nl,PAS_P_ENT)
               END
{-------------------------------------------------------------------
  Pas de '.' après entier
-------------------------------------------------------------------}
            ELSE
               BEGIN
                  liresymb;
                  IF NOT ( symb.ibsymb=specsymb )
                     THEN
                  BEGIN
                     sxerreur:=TRUE;
                     IF symb.ibsymb=errsymb
                        THEN affichage(nl,SYMB_ERR)
                     ELSE
                        IF symb.ibsymb=nosymb
                           THEN affichage(nl,PAS_SYMB)
                        ELSE affichage(nl,PAS_SS_P)
                  END
{-------------------------------------------------------------------
  Pas de specsymb après le premier '.'
-------------------------------------------------------------------}
                  ELSE
                     IF NOT( symb.cval='.' )
                        THEN
                     BEGIN
                        sxerreur:=TRUE;
                        affichage(nl,PAS_P_PREM)
                     END
{-------------------------------------------------------------------
  Pas de '.' après le premier '..
-------------------------------------------------------------------}
                     ELSE
                        ch_tabvar
               END
         END
      END
```

```
  END;{*tab_var*}


{*************************************************************************}
{*************************************************************************}

BEGIN {*sxvar*}

  cour := TRUE;
  new(point_con) ;
  point_con^.num_ligne:=ligne;
  IF NOT ( symb.ibsymb=identificateur )
    THEN
  BEGIN
    sxerreur:=TRUE;
    IF symb.ibsymb=errsymb
      THEN affichage(nl,SYMB_ERR)
    ELSE
      IF symb.ibsymb=nosymb
        THEN affichage(nl,PAS_SYMB)
      ELSE affichage(nl,PAS_ID_VAR)
  END
{-------------------------------------------------------------------
  Pas d'identificateur après VAR
-------------------------------------------------------------------}
  ELSE
  BEGIN
    tete:=NIL;
    nouveau:=NIL;
    REPEAT
      precedent:=nouveau;
      new(nouveau);
      nouveau^.lsident:=NIL;
      nouveau^.pident:=symb.cval;
      nouveau^.num_ligne:=ligne;
      IF tete=NIL
        THEN tete:=nouveau
      ELSE precedent^.lsident:=nouveau;
      liresymb;
{-------------------------------------------------------------------
  Lecture de ',' ou ':' après l'identificateur
-------------------------------------------------------------------}
      IF ( symb.ibsymb=specsymb )
        THEN
      BEGIN
        IF ( symb.cval=',')
          THEN
        BEGIN
          cour  := FALSE;
          liresymb;
          IF NOT( symb.ibsymb = identificateur )
            THEN
          BEGIN
            sxerreur:=TRUE;
            IF symb.ibsymb=errsymb
              THEN affichage(nl,SYMB_ERR)
            ELSE
              IF symb.ibsymb=nosymb
```

```
                    THEN affichage(nl,PAS_SYMB)
                  ELSE affichage(nl,PAS_ID_V)
              END
{-------------------------------------------------------------------------
  Pas d'identificateur après ','
-------------------------------------------------------------------------}
          END
          ELSE
            cour := TRUE
        END
        ELSE
        BEGIN
          sxerreur:=TRUE;
          IF symb.ibsymb=errsymb
            THEN affichage(nl,SYMB_ERR)
          ELSE
            IF symb.ibsymb=nosymb
              THEN affichage(nl,PAS_SYMB)
            ELSE affichage(nl,PAS_SS_ID)
        END
{-------------------------------------------------------------------------
  Pas de specsymbole après le dernier identificateur
-------------------------------------------------------------------------}
    UNTIL cour;

    IF NOT( symb.cval=':')
      THEN
    BEGIN
      sxerreur:=TRUE;
      affichage(nl,PAS_DP_ID)
    END
{-------------------------------------------------------------------------
  Pas de : après le dernier identificateur
-------------------------------------------------------------------------}
    ELSE
    BEGIN
{-------------------------------------------------------------------------
  Lecture du type
-------------------------------------------------------------------------}
        liresymb;
        IF NOT( symb.ibsymb=specsymb )
          THEN
        BEGIN
          sxerreur:=TRUE;
          IF symb.ibsymb=errsymb
            THEN affichage(nl,SYMB_ERR)
          ELSE
            IF symb.ibsymb=nosymb
              THEN affichage(nl,PAS_SYMB)
            ELSE affichage(nl,PAS_SS_DP)
        END
{-------------------------------------------------------------------------
   Pas de specsymb après :
-------------------------------------------------------------------------}
        ELSE
          IF NOT (( symb.cval = 'CHAR'    ) OR
            ( symb.cval = 'BOOLEAN' ) OR
            ( symb.cval = 'INTEGER' ) OR
```

```
            ( symb.cval = 'ARRAY'  ))
            THEN
          BEGIN
            sxerreur:=TRUE;
            affichage(nl,PAS_CBI_DP)
          END
{-------------------------------------------------------------------------
  Problème dans la déclaration de variables
  ou mauvaise déclaration du tableau
-------------------------------------------------------------------------}
          ELSE
            IF (( symb.cval = 'CHAR'   ) OR
              ( symb.cval = 'BOOLEAN' ) OR
              ( symb.cval = 'INTEGER' ))
              THEN
            BEGIN
              point_con^.iconstr:=vari;
              IF symb.cval='CHAR'
                 THEN
                 point_con^.vartype:=caractere;
              IF symb.cval='BOOLEAN'
                 THEN
                 point_con^.vartype:=booleen;
              IF symb.cval='INTEGER'
                 THEN
                 point_con^.vartype:=entier;
              point_con^.lvnom:=tete;
              liresymb;
              sxvar:=point_con
            END
            ELSE
              tab_var
    END
  END
END; {*sxvar*}
{-------------------------------------------------------------------------
  Construit la représentation interne des déclarations d'expressions
  conditionnelles à partir de la représentation externe
-------------------------------------------------------------------------}
FUNCTION sxfoncif: ty_constr;
  VAR
    point_con : ty_constr;

BEGIN{*sxfoncif*}
  IF(symb.ibsymb<> errsymb) AND (symb.ibsymb<> nosymb)
    THEN
  BEGIN
    IF symb.cval= 'IF'
      THEN
    BEGIN
      point_con^.iconstr:= cond2;
      liresymb;
      point_con^.expr1:=sxexpr;
      IF symb.ibsymb=specsymb
        THEN
        IF symb.cval='THEN'
          THEN
        BEGIN
```

```
            liresymb;
            point_con^.instr1:=sxfoncif;
            IF symb.ibsymb=specsymb
               THEN
               IF symb.cval='ELSE'
                  THEN
               BEGIN
                  liresymb;
                  point_con^.instr2:=sxfoncif
               END
               ELSE
               BEGIN
                  point_con^.iconstr:=cond1;
                  point_con^.instr2:=NIL
               END;
            sxfoncif:=point_con
         END
         ELSE sxerreur:=TRUE
         ELSE sxerreur:=TRUE;
      IF sxerreur=TRUE
         THEN affichage(nl,THEN_ATTENDU)
   END
  END
  ELSE
  BEGIN
    sxerreur:=TRUE;
    IF symb.ibsymb=errsymb
       THEN affichage(nl,SYMB_PAS_TR)
    ELSE affichage(nl,FIN_FICH_ERR)
  END
END{*sxfoncif*};

{-------------------------------------------------------------------
  Construit la représentation interne des ensembles d'inéquations
  à partir de la représentation externe
-------------------------------------------------------------------}
FUNCTION sxineq:ty_constr;

  VAR
    point_con :ty_constr;
    nouveau   :ty_lconstr;
    precedent :ty_lconstr;
    tete      :ty_lconstr;
BEGIN{*sxineq*}
  IF (symb.ibsymb<>errsymb) AND (symb.ibsymb<>nosymb)
    THEN
  BEGIN
    IF symb.cval='BEGIN'
       THEN
    BEGIN
{-------------------------------------------------------------------
  Construit la représentation interne du début de programme
-------------------------------------------------------------------}
       cour:=TRUE;
       new(point_con);
       point_con^.num_ligne:=ligne;
       point_con^.iconstr:=debut;
       point_con^.lineq:=NIL;
```

```
        liresymb;
      END;

      IF (symb.ibsymb IN [identificateur,caractere,
        booleen,entier]) OR
        ((symb.ibsymb=specsymb) AND(symb.cval<>'END') AND
        (symb.cval<>'.'))
        THEN
      BEGIN
        tete:=NIL;
        nouveau:=NIL;
        REPEAT
{-------------------------------------------------------------------------
  Construit la représentation interne des inéquations
-------------------------------------------------------------------------}
          precedent:=nouveau;
          new(nouveau);
          nouveau^.lsconstr:=NIL;
          nouveau^.pconstr:=sxexpr;
          IF tete=NIL
            THEN tete:=nouveau
          ELSE precedent^.lsconstr:=nouveau;
          IF (symb.cval=';')
            THEN
          BEGIN
            cour:=FALSE;
            liresymb;
            IF symb.cval='END'
              THEN cour:=TRUE;
          END
          ELSE
            cour:=TRUE
        UNTIL cour;
      END ;
      IF symb.cval='END'
        THEN
      BEGIN
        liresymb;
        point_con^.lineq:=tete;
        sxineq:=point_con
      END
      ELSE sxerreur:=TRUE;
      IF sxerreur=TRUE THEN
        affichage(nl,END_ATTENDU)
    END
    ELSE
    BEGIN
      sxerreur:=TRUE;
      IF symb.ibsymb=errsymb
        THEN affichage(nl,SYMB_PAS_TR)
      ELSE affichage(nl,FIN_FICH_ERR)
    END
  END{*sxineq*};

  {-------------------------------------------------------------------------
    Construit la représentation interne de catégorie
    "facteur" à partir de la représentation externe
  -------------------------------------------------------------------------}
```

```
FUNCTION sxfacteur:ty_constr;

  VAR
    point_con   :ty_constr;
    val_enreg   :chaine;
    pointeur    :ty_lconstr;
    nouveau     :ty_lconstr;
    precedent   :ty_lconstr;

BEGIN {*sxfacteur*}
  IF (symb.ibsymb<>errsymb) AND (symb.ibsymb<>nosymb)
    THEN
  BEGIN
    new(point_con);
    point_con^.num_ligne:=ligne;
    IF symb.ibsymb IN [caractere,booleen,entier]
      THEN
    BEGIN
      point_con^.iconstr:=cons;
      CASE symb.ibsymb OF
      caractere: point_con^.constype:=caractere;
      booleen: point_con^.constype:=booleen;
      entier: point_con^.constype:=entier
      END;
      point_con^.consval:=symb.consval;
      liresymb;
      sxfacteur:=point_con
    END
    ELSE
    BEGIN
      IF symb.ibsymb=identificateur
        THEN
      BEGIN
        val_enreg:=symb.cval;
        liresymb;
        IF (symb.ibsymb=specsymb) AND (symb.cval='[')
          THEN
        BEGIN
          point_con^.iconstr:=indvar;
          point_con^.tnom:=val_enreg;
          liresymb;
          point_con^.indice:=symb.consval;
          liresymb;

          IF symb.ibsymb=specsymb
            THEN
            IF symb.cval=']'
              THEN
            BEGIN
              liresymb;
              sxfacteur:=point_con
            END
            ELSE sxerreur:=TRUE
            ELSE sxerreur:=TRUE;
          IF sxerreur=TRUE
            THEN affichage(nl,CF_ATTENDU)
        END
        ELSE
```

```
            IF (symb.ibsymb=specsymb) AND (symb.cval='(')
               THEN
            BEGIN
               point_con^.iconstr:=fonction;
               point_con^.fnom:=val_enreg;
               point_con^.leff:=NIL;
               nouveau:=NIL;
               REPEAT
                  precedent:=nouveau;
                  new(nouveau);
                  liresymb;
                  nouveau^.pconstr:=sxexpr;
                  nouveau^.lsconstr:=NIL;
                  IF point_con^.leff=NIL
                     THEN point_con^.leff:=nouveau
                  ELSE precedent^.lsconstr:=nouveau
               UNTIL ((symb.ibsymb<>specsymb) OR
               (symb.cval<>','));
               IF symb.ibsymb=specsymb
                  THEN
                  IF symb.cval=')'
                     THEN
                  BEGIN
                     liresymb;
                     sxfacteur:=point_con
                  END
                  ELSE sxerreur:=TRUE
                  ELSE sxerreur:=TRUE;
               IF sxerreur=TRUE
                  THEN affichage(nl,PF_ATTENDU)
            END
            ELSE
            BEGIN
               point_con^.iconstr:=simvar;
               point_con^.vnom:=val_enreg;
               sxfacteur:=point_con
            END
         END
         ELSE
           IF (symb.ibsymb=specsymb) AND (symb.cval='(')
              THEN
           BEGIN
              liresymb;
              sxfacteur:=sxexpr;
              IF symb.ibsymb=specsymb
                 THEN
                 IF symb.cval=')'
                    THEN liresymb
                 ELSE sxerreur:=TRUE
                 ELSE sxerreur:=TRUE;
              IF sxerreur=TRUE
                 THEN affichage(nl,PF_ATTENDU)
           END
           ELSE
{-------------------------------------------------------------------------
  Dans ce cas, la construction présentée à la fonction
  n'appartient pas à la catégorie "facteur"
-------------------------------------------------------------------------}
```

```
          BEGIN
            sxerreur:=TRUE;
            affichage(nl,SYMB_FACT_ERR)
          END
      END
    END
    ELSE
    BEGIN
      sxerreur:=TRUE;
      IF symb.ibsymb=errsymb
        THEN affichage(nl,SYMB_PAS_TR)
      ELSE affichage(nl,FIN_FICH_ERR)
    END
END; {*sxfacteur*}

{-------------------------------------------------------------------
  Construit la représentation interne de catégorie
  "terme" à partir de la représentation externe
-------------------------------------------------------------------}
FUNCTION sxterme:ty_constr;

  VAR
    point_con:ty_constr;
    aid_pont:ty_constr;

BEGIN {*sxterme*}
  IF (symb.ibsymb<>errsymb) AND (symb.ibsymb<>nosymb)
    THEN
  BEGIN
    point_con:=sxfacteur;
    WHILE (symb.ibsymb=specsymb) AND ((symb.cval='*') OR
      (symb.cval='DIV') OR (symb.cval='MOD'))
      DO
    BEGIN
      new(aid_pont);
      aid_pont^.num_ligne:=ligne;
      aid_pont^.iconstr:=binop;
      IF symb.cval='*'
        THEN aid_pont^.bop:=fois
      ELSE
        IF symb.cval='DIV'
          THEN aid_pont^.bop:=quotient
        ELSE
          IF symb.cval='MOD'
            THEN aid_pont^.bop:=reste;
      aid_pont^.oper1:=point_con;
      liresymb;
      aid_pont^.oper2:=sxfacteur;
{-------------------------------------------------------------------
  Construit la représentation interne de l'opérateur multiplicatif
  en s'arrangeant pour que point_con ait la valeur du pointeur
  sur cette structure
-------------------------------------------------------------------}
      point_con:=aid_pont
    END;
    sxterme:=point_con
  END
  ELSE
```

```
  BEGIN
    sxerreur:=TRUE;
    IF symb.ibsymb=errsymb
      THEN affichage(nl,SYMB_PAS_TR)
    ELSE affichage(nl,FIN_FICH_ERR)
  END
END; {*sxterme*}

{-------------------------------------------------------------------------
  Construit la représentation interne de catégorie
  "aexpr" à partir de la représentation externe
-------------------------------------------------------------------------}
FUNCTION sxaexpr:ty_constr;

  VAR
    unop_pont  :ty_constr;
    point_con  :ty_constr;
    aid_pont   :ty_constr;

BEGIN {*sxaexpr*}
  IF (symb.ibsymb<>errsymb) AND (symb.ibsymb<>nosymb)
    THEN
  BEGIN
    IF symb.ibsymb=specsymb
      THEN
      IF symb.cval='-'
        THEN
      BEGIN
        new(unop_pont);
        unop_pont^.num_ligne:=ligne;
        unop_pont^.iconstr:=unop;
        unop_pont^.uop:=moins;
        liresymb;
        unop_pont^.oper:=sxterme;
        point_con:=unop_pont
      END
      ELSE point_con:=sxterme
      ELSE point_con:=sxterme;
    WHILE (symb.ibsymb=specsymb) AND ((symb.cval='+') OR
      (symb.cval='-'))
      DO
    BEGIN
      new(aid_pont);
      aid_pont^.num_ligne:=ligne;
      aid_pont^.iconstr:=binop;
      IF symb.cval='+'
        THEN aid_pont^.bop:=plus
      ELSE
        IF symb.cval='-'
          THEN aid_pont^.bop:=moins;
      aid_pont^.oper1:=point_con;
      liresymb;
      aid_pont^.oper2:=sxterme;
{-------------------------------------------------------------------------
  Construit la représentation interne de l'expression arithmétique
  en s'arrangeant pour que point_con ait la valeur du pointeur
  sur cette structure
-------------------------------------------------------------------------}
```

```
        point_con:=aid_pont
      END;
      sxaexpr:=point_con
    END
    ELSE
    BEGIN
      sxerreur:=TRUE;
      IF symb.ibsymb=errsymb
        THEN affichage(nl,SYMB_PAS_TR)
      ELSE affichage(nl,FIN_FICH_ERR)
    END
END; {*sxaexpr*}

{-------------------------------------------------------------------------
  Construit la représentation interne de catégorie
  "négation" à partir de la représentation externe
-------------------------------------------------------------------------}
FUNCTION sxneg:ty_constr;

  VAR
    neg         :BOOLEAN;
    unop_pont   :ty_constr;
    point_con   :ty_constr;
    aid_pont    :ty_constr;
    binop_point :ty_constr;

BEGIN {*sxneg*}
  IF (symb.ibsymb<>errsymb) AND (symb.ibsymb<>nosymb)
    THEN
  BEGIN
    neg:=FALSE;
    IF symb.ibsymb=specsymb
      THEN
      IF symb.cval='NOT'
        THEN
      BEGIN
        neg:=TRUE;
        new(unop_pont);
        unop_pont^.num_ligne:=ligne;
        unop_pont^.iconstr:=unop;
        unop_pont^.uop:=non;
        liresymb;
        unop_pont^.oper:=sxaexpr;
        point_con:=unop_pont
      END
      ELSE point_con:=sxaexpr
      ELSE point_con:=sxaexpr;
    IF symb.ibsymb=specsymb
      THEN
      IF (symb.cval='<') OR (symb.cval='>')
        OR (symb.cval='=')
        THEN
        IF NOT neg
          THEN
        BEGIN
          new(aid_pont);
          aid_pont^.num_ligne:=ligne;
          aid_pont^.iconstr:=binop;
```

```
            IF symb.cval='<'
               THEN
            BEGIN
               liresymb;
              IF symb.ibsymb=specsymb
                 THEN
                 IF (symb.cval='>')
                   OR (symb.cval='=')
                    THEN
                 BEGIN
                   IF symb.cval='>'
                      THEN aid_pont^.bop:=different
                   ELSE aid_pont^.bop:=infeq;
                   liresymb
                 END
                 ELSE aid_pont^.bop:=inferieur
                 ELSE aid_pont^.bop:=inferieur
            END
            ELSE
              IF symb.cval='>'
                 THEN
              BEGIN
                 liresymb;
                IF symb.ibsymb=specsymb
                   THEN
                   IF symb.cval='='
                      THEN
                   BEGIN
                     aid_pont^.bop:=supeq;
                     liresymb
                   END
                   ELSE aid_pont^.bop:=superieur
                   ELSE aid_pont^.bop:=superieur
              END
              ELSE
                IF symb.cval='='
                   THEN
                BEGIN
                   aid_pont^.bop:=egal;
                   liresymb
                END;
            aid_pont^.oper1:=point_con;
            aid_pont^.oper2:=sxaexpr;
{-------------------------------------------------------------------------
  Construit la représentation interne de l'expression relationnelle
  en s'arrangeant pour que point_con pointe sur cette structure
-------------------------------------------------------------------------}
              point_con:=aid_pont
          END
          ELSE
          BEGIN
            new(binop_point);
            binop_point^.num_ligne:=ligne;
            binop_point^.iconstr:=binop;
            IF symb.cval='<'
               THEN
            BEGIN
               liresymb;
```

```
                IF symb.ibsymb=specsymb
                   THEN
                   IF (symb.cval='>')
                      OR (symb.cval='=')
                      THEN
                   BEGIN
                      IF symb.cval='>'
                         THEN binop_point^.bop:=different
                      ELSE binop_point^.bop:=infeq;
                      liresymb
                   END
                   ELSE binop_point^.bop:=inferieur
                   ELSE binop_point^.bop:=inferieur
                END
                ELSE
                  IF symb.cval='>'
                     THEN
                  BEGIN
                     liresymb;
                     IF symb.ibsymb=specsymb
                        THEN
                        IF symb.cval='='
                           THEN
                        BEGIN
                           binop_point^.bop:=supeq;
                           liresymb
                        END
                        ELSE binop_point^.bop:=superieur
                        ELSE binop_point^.bop:=superieur
                  END
                  ELSE
                     IF symb.cval='='
                        THEN
                     BEGIN
                        binop_point^.bop:=egal;
                        liresymb
                     END;
                binop_point^.oper1:=point_con^.oper;
                point_con^.oper:=binop_point;
                binop_point^.oper2:=sxaexpr
             END;
          sxneg:=point_con
       END
       ELSE
       BEGIN
          sxerreur:=TRUE;
          IF symb.ibsymb=errsymb
             THEN affichage(nl,SYMB_PAS_TR)
          ELSE affichage(nl,FIN_FICH_ERR)
       END
    END; {*sxneg*}

    {-----------------------------------------------------------------------
       Construit la représentation interne de catégorie
       "conjonction" à partir de la représentation externe
    -----------------------------------------------------------------------}
    FUNCTION sxconj:ty_constr;
```

```
  VAR
    point_con   :ty_constr;
    aid_pont    :ty_constr;

BEGIN {*sxconj*}
  IF (symb.ibsymb<>errsymb) AND (symb.ibsymb<>nosymb)
    THEN
  BEGIN
    point_con:=sxneg;
    IF symb.ibsymb=specsymb
      THEN
      IF symb.cval='AND'
        THEN
      BEGIN
        new(aid_pont);
        aid_pont^.num_ligne:=ligne;
        aid_pont^.iconstr:=binop;
        aid_pont^.bop:=et;
        aid_pont^.oper1:=point_con;
        liresymb;
        aid_pont^.oper2:=sxconj;
{-------------------------------------------------------------------------
  Construit la représentation interne du binop "and"
  en s'arrangeant pour que point_con pointe sur cette structure
-------------------------------------------------------------------------}
        point_con:=aid_pont
      END;
    sxconj:=point_con
  END
  ELSE
  BEGIN
    sxerreur:=TRUE;
    IF symb.ibsymb=errsymb
      THEN affichage(nl,SYMB_PAS_TR)
    ELSE affichage(nl,FIN_FICH_ERR)
  END
END; {*sxconj*}

{-------------------------------------------------------------------------
  Construit la représentation interne de catégorie
  "expression" à partir de la représentation externe
-------------------------------------------------------------------------}
FUNCTION sxexpr:ty_constr;

  VAR
    point_con   :ty_constr;
    aid_pont    :ty_constr;

BEGIN {*sxexpr*}
  IF (symb.ibsymb<>errsymb) AND (symb.ibsymb<>nosymb)
    THEN
  BEGIN
    point_con:=sxconj;
    IF symb.ibsymb=specsymb
      THEN
      IF symb.cval='OR'
        THEN
      BEGIN
```

```
            new(aid_pont);
            aid_pont^.num_ligne:=ligne;
            aid_pont^.iconstr:=binop;
            aid_pont^.bop:=ou;
            aid_pont^.oper1:=point_con;
            liresymb;
            aid_pont^.oper2:=sxexpr;
{-------------------------------------------------------------------
  Construit la représentation interne du binop "or"
  en s'arrangeant pour que point_con pointe sur cette structure
-------------------------------------------------------------------}
            point_con:=aid_pont
          END;
        sxexpr:=point_con
    END
    ELSE
    BEGIN
      sxerreur:=TRUE;
      IF symb.ibsymb=errsymb
        THEN affichage(nl,SYMB_PAS_TR)
      ELSE affichage(nl,FIN_FICH_ERR)
    END
END; {*sxexpr*}

{-------------------------------------------------------------------
  Construit la représentation interne d'un programme SYSTINEQ
  à partir de la représentation externe décrite dans le fichier de texte
-------------------------------------------------------------------}
FUNCTION sxprog;
  VAR
    sxerreur    :BOOLEAN;
    nouveau     :ty_lconstr;
    precedent   :ty_lconstr;
    avance      :BOOLEAN;
    cour        :BOOLEAN;
    nl          :INTEGER;
BEGIN {*sxprog*}
          {*initialisation*}
  sxerreur:=FALSE;
  ligne := 1 ;
  new( ptr_prg );
  caract_pres := FALSE;
{-------------------------------------------------------------------
  Ici commence la lecture du fichier d'entrée
-------------------------------------------------------------------}
  liresymb;

  IF  NOT( symb.ibsymb = specsymb )
    THEN
  BEGIN
    sxerreur := TRUE;
    IF symb.ibsymb = errsymb
      THEN affichage(nl,SYMB_ERR)
    ELSE
      IF symb.ibsymb=nosymb
        THEN affichage(nl,PAS_SYMB)
      ELSE affichage(nl,PAS_SS_PREM)
{-------------------------------------------------------------------
```

```
  Pas de specsymbole en première position
-------------------------------------------------------------------------}
  END
  ELSE
    IF NOT( symb.cval='PROGRAM' )
      THEN
    BEGIN
      sxerreur:=TRUE;
      affichage(nl,PAS_PROG_SS)
{-------------------------------------------------------------------------
  Pas de chaîne PROGRAM comme specsymbole
-------------------------------------------------------------------------}
    END
    ELSE
    BEGIN
      ptr_prg^.iconstr:=prog;
      liresymb;
      IF NOT( symb.ibsymb=identificateur )
        THEN
      BEGIN
        sxerreur:=TRUE;
        IF symb.ibsymb=errsymb
          THEN affichage(nl,SYMB_ERR)
        ELSE
          IF symb.ibsymb=nosymb
            THEN affichage(nl,PAS_SYMB)
          ELSE affichage(nl,PAS_ID_PROG)
{-------------------------------------------------------------------------
  Pas d'identificateur après la chaîne PROGRAM
-------------------------------------------------------------------------}
      END
      ELSE
      BEGIN
{-------------------------------------------------------------------------
  Construction interne de la déclaration du nom de programme
-------------------------------------------------------------------------}
        ptr_prg^.pnom:=symb.cval;
        ptr_prg^.lvar:=NIL;
        liresymb;
        IF NOT( symb.ibsymb=specsymb )
          THEN
        BEGIN
          sxerreur:=TRUE;
          IF symb.ibsymb=errsymb
            THEN affichage(nl,SYMB_ERR)
          ELSE
            IF symb.ibsymb=nosymb
              THEN affichage(nl,PAS_SYMB)
            ELSE affichage(nl,PAS_SS_ID)
{-------------------------------------------------------------------------
  Pas de specsymbole après l'identificateur
-------------------------------------------------------------------------}
        END
        ELSE
          IF NOT( symb.cval=';' )
            THEN
          BEGIN
            sxerreur:=TRUE;
```

```
                affichage(nl,PAS_PV_ID)
              END
{-------------------------------------------------------------------------
  Pas de ';' après l'identificateur
-------------------------------------------------------------------------}
              ELSE

              BEGIN
{-------------------------------------------------------------------------
  Après la déclaration du nom de programme
-------------------------------------------------------------------------}
                liresymb;
                IF NOT( symb.ibsymb = specsymb )
                   THEN
                BEGIN
                   sxerreur := TRUE;
                   IF   symb.ibsymb = errsymb
                      THEN affichage(nl,SYMB_ERR);
                   IF symb.ibsymb = nosymb
                      THEN affichage(nl,PAS_SYMB);
                   affichage(nl,PAS_SS_PV)
                END;
{-------------------------------------------------------------------------
  Pas de specsymbole après ';'
-------------------------------------------------------------------------}

                IF NOT( ( symb.cval =   'VAR'   ) OR
                   ( symb.cval =   'BEGIN' ) )
                   THEN
                BEGIN
                   sxerreur := TRUE;
                   affichage(nl,PAS_VAR_PV)
{-------------------------------------------------------------------------
  Pas le bon specsymbole
-------------------------------------------------------------------------}
                END ;
{-------------------------------------------------------------------------
  Construction de la représentation interne
  des déclarations de variables
-------------------------------------------------------------------------}
                IF NOT( symb.cval = 'VAR' )
                   THEN ptr_prg^.lvar := NIL
                ELSE
                BEGIN
                   avance          := TRUE;
                   ptr_prg^.lvar := NIL;
                   nouveau      := NIL;
                   liresymb;
{-------------------------------------------------------------------------
  On a passé la chaîne VAR
-------------------------------------------------------------------------}
                   REPEAT
                      precedent := nouveau ;
                      new( nouveau ) ;
                      nouveau^.pconstr := sxvar ;
                      nouveau^.lsconstr := NIL ;
                      IF   ptr_prg^.lvar = NIL
                         THEN ptr_prg^.lvar := nouveau
```

```
                    ELSE precedent^.lsconstr := nouveau;
                    IF NOT( symb.ibsymb = specsymb )
                       THEN
                    BEGIN
                       sxerreur := TRUE;
                       IF symb.ibsymb = errsymb
                          THEN affichage(nl,SYMB_ERR);
                       IF symb.ibsymb = nosymb
                          THEN affichage(nl,PAS_SYMB);
                       affichage(nl,PAS_SS_DV)
                    END;
{-------------------------------------------------------------------------
  Pas de specsymbole après les déclarations de variables
-------------------------------------------------------------------------}
                    IF NOT ( symb.cval = ';')
                       THEN
                    BEGIN
                       sxerreur := TRUE;
                       affichage(nl,PAS_PV_DV)
                    END;
{-------------------------------------------------------------------------
  Pas de ';' après la déclaration de variables
-------------------------------------------------------------------------}
                    liresymb;
                    IF NOT( ( symb.ibsymb = specsymb )  OR
                       ( symb.ibsymb = identificateur ) )
                       THEN
                    BEGIN
                       sxerreur := TRUE;
                       IF symb.ibsymb = errsymb
                          THEN affichage(nl,SYMB_ERR);
                       IF symb.ibsymb = nosymb
                          THEN affichage(nl,PAS_SYMB);
                       affichage(nl,PAS_SSID_PV)
                    END;
{-------------------------------------------------------------------------
  Pas de specsymbole ou d'identificateur après la déclaration
  de variables
-------------------------------------------------------------------------}
                    IF symb.ibsymb = specsymb
                       THEN
                    BEGIN
                       IF (   symb.cval =   'BEGIN' )
                          THEN  avance := FALSE
                       ELSE
                       BEGIN
                          sxerreur := TRUE;
                          affichage(nl,PAS_BEG_PV)
                       END
{-------------------------------------------------------------------------
  Pas de BEGIN comme specsymbole après ';'
-------------------------------------------------------------------------}
                    END
                 UNTIL NOT( ( symb.ibsymb = identificateur) OR
                 ( avance ) )
              END;
```

```
{-------------------------------------------------------------------------
  Construction de la représentation interne d'une
  suite d'inéquations
-------------------------------------------------------------------------}
                ptr_prg^.corps := sxineq ;
                IF NOT( symb.ibsymb = specsymb )
                   THEN
                BEGIN
                   sxerreur := TRUE;
                   IF symb.ibsymb = errsymb
                      THEN affichage(nl,SYMB_ERR);
                   IF symb.ibsymb = nosymb
                      THEN affichage(nl,PAS_SYMB);
                   affichage(nl,PAS_SS_ENDF)
                END;
{-------------------------------------------------------------------------
  Pas de specsymb après END
-------------------------------------------------------------------------}
                IF NOT ( symb.cval = '.')
                   THEN
                BEGIN
                   sxerreur := TRUE;
                   affichage(nl, PAS_P_END)
                END;{*Pas de "." après end *}
                liresymb;
                IF NOT( symb.ibsymb = nosymb )
                   THEN
                BEGIN
                   sxerreur := TRUE;
                   affichage(nl, PAS_Q_P)
                END
{-------------------------------------------------------------------------
  Pas de '?' après '.'
-------------------------------------------------------------------------}
             END;
          END;
       END ;
    sxprog := ptr_prg ;
END;{*sxprog*}

END.{*unit*}
```

fichier BiblFonc.pas (la bibliothèque de fonctions disponibles à utiliser par les algorithmes de calcul du point fixe)

```
UNIT BiblFonc ;

INTERFACE
USES DefGlob;

CONST

   NbFon=17;
```

```
TYPE
  ty_fonc =  RECORD
                cval          : chaine;
                ty_oper       : operateur;
                ftype : valtype;
                nbr_arg       : INTEGER;
             END;

  table_fonc=ARRAY[1..NbFon] OF ty_fonc;

VAR
  tab_fonc:table_fonc;

procedure ConstrTabFonc;

function ConsultTabFonc(x:ty_constr):integer;

  implementation
{-----------------------------------------------------------------
 Remplit la table des fonctions
 avec 17 fonctions de la bibliothèque
-----------------------------------------------------------------}
PROCEDURE ConstrTabFonc;
BEGIN
  tab_fonc[1].cval:='*';        tab_fonc[1].ty_oper:=fois;
  tab_fonc[2].cval:='div';      tab_fonc[2].ty_oper:=quotient;
  tab_fonc[3].cval:='mod';      tab_fonc[3].ty_oper:=modulo;
  tab_fonc[4].cval:='+';        tab_fonc[4].ty_oper:=plus;
  tab_fonc[5].cval:='-';        tab_fonc[5].ty_oper:=moins;
  tab_fonc[6].cval:='=';        tab_fonc[6].ty_oper:=egal;
  tab_fonc[7].cval:='<>';       tab_fonc[7].ty_oper:=different;
  tab_fonc[8].cval:='<';        tab_fonc[8].ty_oper:=inferieur;
  tab_fonc[9].cval:='>';        tab_fonc[9].ty_oper:=superieur;
  tab_fonc[10].cval:='<=';      tab_fonc[10].ty_oper:=infeq;
  tab_fonc[11].cval:='>=';      tab_fonc[11].ty_oper:=supeq;
  tab_fonc[12].cval:='and';     tab_fonc[12].ty_oper:=et;
  tab_fonc[13].cval:='or';      tab_fonc[13].ty_oper:=ou;
  tab_fonc[14].cval:='not';     tab_fonc[14].ty_oper:=non;
  tab_fonc[15].cval:='MAX';  tab_fonc[15].ty_oper:=maximum;
  tab_fonc[15].nbr_arg:=2;
  tab_fonc[16].cval:='MIN';   tab_fonc[16].ty_oper:=minimum;
  tab_fonc[16].nbr_arg:=2;
  tab_fonc[17].cval:='COMP'  ;tab_fonc[17].ty_oper:=compar;
  tab_fonc[17].nbr_arg:=1;
END;{ConstrTabFonc}

{-----------------------------------------------------------------
  Renvoie le numéro de fonction si la fonction est présente
  dans la table; sinon, un message d'erreur
-----------------------------------------------------------------}
FUNCTION ConsultTabFonc( x: ty_constr): INTEGER;
  VAR
    num_l,i     :INTEGER;
    absence     :BOOLEAN;

BEGIN{ConsultTabFonc}
  absence:=TRUE;
```

```
  FOR i:=1 TO NbFon DO
    IF ((x^.bop=tab_fonc[i].ty_oper) OR
      (x^.uop=tab_fonc[i].ty_oper) OR
      (x^.fnom=tab_fonc[i].cval))
      THEN
    BEGIN
      ConsultTabFonc:=i ;
      absence:=FALSE;
    END;
  IF absence THEN
    affichage(num_l,PAS_IN_BIBLIO)
END{ConsultTabFonc};
END.{unit}
```

fichier CrPile.pas (les procédures et les fonctions de constructeur de la pile d'opérateurs et d'opérandes

```
UNIT crpile;

INTERFACE
USES DefGlob,BiblFonc;

CONST
  NbVar=300;
  NbFIneq=1500;

TYPE

  ty_var = RECORD
             iconstr      :tconstr;
             vnom         :chaine;
             vtype        :valtype;
             valeur       :INTEGER;
             indice       :INTEGER
           END;
  ty_tab_var=ARRAY[0..NbVar] OF ty_var;

  elem_tab_ineq = RECORD
    valeur      :INTEGER;
    CASE iconstr: tconstr OF
    simvar,indvar:          ( num_var   :INTEGER;
                              vartype   :valtype;
                              ind       :INTEGER);
    unop,binop,fonction:    ( num_oper  :INTEGER;
                              optype    :valtype;
                              nbr_arg   :INTEGER);
    cons:                   (constype   :valtype;
                              consval   :INTEGER);
                    END;
  ty_tab_in =ARRAY[0..NbFIneq] OF elem_tab_ineq;

  ty_arg=^el_arg;
  el_arg= RECORD
```

```
            n_arg       : INTEGER;
            arg_suiv    : ty_arg;
        END;

VAR

  NbVarCour,NbEl   :INTEGER;
  tab_var          :ty_tab_var;
  pile             :ty_tab_in;
  value            :ARRAY[1..NbVar] OF INTEGER;

procedure ConstrPileIneq(prg:ty_constr);
procedure ConstrTabVar(x:ty_constr;var NbVarCour:integer);


  implementation

VAR
  nl          :INTEGER;


procedure PileIneq(n:integer;x:ty_constr;var num:integer)    ;forward;

{-----------------------------------------------------------------
  Construction de la table des variables à partir de la
  représentation interne d'un programme
-----------------------------------------------------------------}
PROCEDURE ConstrTabVar(x:ty_constr; VAR NbVarCour:INTEGER);

  VAR
    dec_var    :ty_lconstr;
    aux1       :ty_constr;
    aux2       :ty_lident;
    i,j        :INTEGER;

BEGIN{ConstrTabVar}
  tab_var[0].vnom :='!'; {*Elément de séparation entre des inéquations*}
  tab_var[0].vtype:=caractere;

  i:=1;
  dec_var:=x^.lvar;
{-----------------------------------------------------------------
  Pointeur vers la tête des déclarations de variables
-----------------------------------------------------------------}
  WHILE (dec_var <>NIL) DO
  BEGIN
    CASE dec_var^.pconstr^.iconstr OF
    tab:        BEGIN
                aux1:=dec_var^.pconstr;
                 WHILE aux1^.ltnom <> NIL DO
                     BEGIN
                     FOR j:=aux1^.db TO aux1^.fb DO
                     BEGIN
                             tab_var[i].iconstr:=tab;
                             tab_var[i].vnom:=aux1^.ltnom^.pident;
                             tab_var[i].vtype:=aux1^.tabtype;
                             tab_var[i].indice:=j;
                             i:=i+1
```

```
                    END;
                    aux1^.ltnom:=aux1^.ltnom^.lsident
                END
                END;
    vari:       BEGIN
                aux2:=dec_var^.pconstr^.lvnom;
                WHILE (aux2<>NIL) DO
                BEGIN
                    tab_var[i].iconstr:=vari;
                    tab_var[i].vtype:= dec_var^.pconstr^.vartype;
                    tab_var[i].vnom:= aux2^.pident;
                    tab_var[i].indice:=-1;
                    i:=i+1;
                    aux2:=aux2^.lsident;
                END
                END
                END;
                dec_var:=dec_var^.lsconstr;
                END;
  NbVarCour:=i-1 ;
END{ConstrTabVar};

{-----------------------------------------------------------------
  Renvoie le numéro de variable trouvé dans la table des variables
-----------------------------------------------------------------}
FUNCTION ConsultTabVar(x:ty_constr):INTEGER;

  VAR
    i           :INTEGER;
    absence     :BOOLEAN;
BEGIN{ConsultTabVar}
  absence:=TRUE;
  i:=1;
  WHILE absence AND (i<=NbVarCour) DO
  BEGIN
    CASE x^.iconstr OF
    simvar:     BEGIN
                IF (x^.vnom= tab_var[i].vnom) THEN
                BEGIN
                    ConsultTabVar:=i;
                    absence:=FALSE;
                END;
                END;
    indvar:     BEGIN
                IF (x^.tnom=tab_var[i].vnom) AND (x^.indice=tab_var[i].indice) THEN
                BEGIN
                    ConsultTabVar:=i;
                    absence:=FALSE
                END;
                END;
                END;
    i:=i+1
  END;

  IF absence THEN
    affichage(nl,PAS_TAB_VAR);
END{ConsultTabVar};
```

```
{-----------------------------------------------------------------
  Remplissage d'une partie de la pile des inéquations à partir
  de la représentation interne d'une inéquation
-----------------------------------------------------------------}
PROCEDURE PileIneq(n:INTEGER; x:ty_constr; VAR num: INTEGER);

  VAR i:INTEGER;
    y:ty_lconstr;

BEGIN{PileIneq}
  CASE x^.iconstr OF

  fonction:     BEGIN
                    y:=x^.leff;
                    WHILE (y<>NIL) DO
                    BEGIN
                       PileIneq(n,y^.pconstr,num);
                       y:=y^.lsconstr;
                    END;
                    i:=ConsultTabFonc(x);
                    pile[num].iconstr:=fonction;
                    pile[num].num_oper:=i;
                    pile[num].nbr_arg:=tab_fonc[i].nbr_arg;
                    pile[num].optype:=tab_fonc[i].ftype;
                    num:=num+1;
                END;
  simvar:       BEGIN
                    i:=ConsultTabVar(x);
                    pile[num].iconstr:=simvar;
                    pile[num].num_var:=i;
                    pile[num].vartype:=tab_var[i].vtype;
                    pile[num].ind:=-1;
                    num:=num+1
                END;
  indvar:       BEGIN
                    i:=ConsultTabVar(x);
                    pile[num].iconstr:=indvar;
                    pile[num].num_var:=i;
                    pile[num].vartype:=tab_var[i].vtype;
                    pile[num].ind:=tab_var[i].indice;
                    num:=num+1
                END;

  cons:         BEGIN
                    pile[num].iconstr:=cons;
                    pile[num].constype:=x^.constype;
                    pile[num].consval:=x^.consval;
                    num:=num+1
                END;

  unop:         BEGIN
                IF (x<> NIL) THEN
                BEGIN
                    PileIneq(n,x^.oper, num);
                    i:=ConsultTabFonc(x);
                    pile[num].iconstr:=unop;
                    pile[num].num_oper:=i;
                    pile[num].optype:=tab_fonc[i].ftype;
```

```
                    pile[num].nbr_arg:=1;
                    num:=num+1
                END
                END;
  binop:        BEGIN
                IF (x<>NIL) THEN
                BEGIN
                    PileIneq(n,x^.oper1,num);
                    PileIneq(n,x^.oper2,num);
                    i:=ConsultTabFonc(x);
                    pile[num].iconstr:=binop;
                    pile[num].num_oper:=i;
                    pile[num].optype:=tab_fonc[i].ftype;
                    pile[num].nbr_arg:=2;
                    num:=num+1
                END
                END
  END
END{PileIneq};

{-----------------------------------------------------------------
  Remplissage de la pile des inéquations à partir
  de la représentation interne des inéquations
-----------------------------------------------------------------}
PROCEDURE ConstrPileIneq(prg:ty_constr);
  VAR
    var_ineq    :ty_lconstr;
    ineq        :ty_constr;
    i,n,numop   :INTEGER;

BEGIN{ConstrPileIneq}
  n:=1;
  numop:=1;
  ConstrTabVar(prg,NbVarCour);
  ConstrTabFonc;
  var_ineq:=prg^.corps^.lineq;
{-----------------------------------------------------------------
  Pointeur vers la représentation interne des inéquations
-----------------------------------------------------------------}
  WHILE (var_ineq<>NIL) DO
  BEGIN
    ineq:=var_ineq^.pconstr;
    PileIneq(n,ineq,numop);
{-----------------------------------------------------------------
  Construction d'une partie de la pile
  correspondant à une inéquation
-----------------------------------------------------------------}
    pile[numop].iconstr:=simvar;{*Separateur entre des inéquations*}
    pile[numop].num_var:=0;

    numop:=numop+1;
    n:=numop ;
    var_ineq:=var_ineq^.lsconstr;
  END;
  NbEl:=numop-1;

  FOR i:=1 TO NbEl DO
  CASE pile[i].iconstr OF
```

```
  fonction:      writeln(fich_out,i,': ', tab_fonc[pile[i].num_oper].cval);

  simvar:        writeln(fich_out,i,': ', tab_var[pile[i].num_var].vnom);
  indva:         writeln(fich_out,i,':' , tab_var[pile[i].num_var].vnom,pile[i].ind);

  unop,binop:    writeln(fich_out,i, ': ', tab_fonc[pile[i].num_oper].cval);

  cons:          writeln(fich_out,i, ': ', pile[i].consval);
  END;

END{ConstrPileIneq};

END{unit}.
```

fichier Algorith.pas (la structure des données utilisées par les algorithmes, les procédures et les fonctions spécifiques aux algorithmes Bottom-Up et Top-Down)

```
UNIT Algorit;

INTERFACE
USES Dos,DefGlob,BiblFonc,CrPile;

procedure Alg1;
procedure Alg2;

implementation

CONST
  LongReg=5;
  bottom=0;

TYPE
{-------------------------------------------------------------------
   Structures de données partagées par les deux algorithmes
-------------------------------------------------------------------}
  ty_borne=RECORD
                debut:      INTEGER;
                fin:        INTEGER
           END;

  chain_var=^el_chain;
  el_chain = RECORD
                num_eq      : INTEGER;
                borne_eq    : ty_borne;
                chsuiv      : chain_var;
             END;
  ty_chain=ARRAY[1.. NbVar] OF chain_var;


  el_reg=RECORD
                val_reg     : INTEGER;
```

```
                libre          : BOOLEAN;
            END;
  ty_reg=ARRAY[1..LongReg] OF el_reg;

{-----------------------------------------------------------------
  Structures de données utilisées par l'algorithme Bottom-Up
-----------------------------------------------------------------}

  ty_stack=^el;
  el= RECORD
                numvar       : INTEGER;
                varsuiv      : ty_stack;
    END;


{-----------------------------------------------------------------
  Structures de données utilisées par l'algorithme Top-Down
-----------------------------------------------------------------}
  el_pt=RECORD
    valeur       : INTEGER;
    num_var      : INTEGER;
        END;
  ty_pt=ARRAY[1..NbVar] OF el_pt;

  el_dg=^suite_dg;
  suite_dg=RECORD
                num_var      :INTEGER;
                el_suiv      :el_dg
            END;
  ty_dg=ARRAY[1..NbVar] OF el_dg;

  ty_optimal=ARRAY[1..NbVar] OF INTEGER;

   ty_ics= ARRAY[1..NbVar] OF BOOLEAN;

VAR
  heure,minute,seconde,centiem_sec      :word;
  h,h1 ,m,m1 ,s,s1 ,c_s,c_s1            :word;
  NbPar,num_eq,num_var,var_dep          :INTEGER;
  ch,chain,chain_gen                    :ty_chain;
  par_g                                 :ARRAY[1..NbVar] OF INTEGER;
  reg                                   :ty_reg;
  nul_dg                                :el_dg;
  compteur,compt_alg1                   :INTEGER;
  suite,dg                              :ty_dg;
  ics                                   :ty_ics;
  pt                                    :ty_pt;
  optimal                               :ty_optimal;
  nouveau,precedent                     :ty_dg;


{*****************************************************************************}
{**************************Procédure calcul ***********************************}
{*****************************************************************************}
{-----------------------------------------------------------------
  Effectue les opérations de calcul proprement dites
  dans la pile des opérateurs et des opérandes
-----------------------------------------------------------------}
```

```
PROCEDURE calcul(pile:ty_tab_in;borne:ty_borne;VAR t:INTEGER);
  VAR
    precedent,tete,nouveau,arg          :ty_arg;
    nbr_arg,i,j,fin,k                   :INTEGER;
    resultat                            :INTEGER;
    trouve                              :BOOLEAN;
    operande                            :ARRAY[1..3] OF INTEGER;

BEGIN
{-------------------------------------------------------------------
  Initialisation
-------------------------------------------------------------------}
  i:=borne.debut;
  FOR j:=1  TO LongReg DO
    reg[j].libre:=TRUE;
{-------------------------------------------------------------------
  Corps
-------------------------------------------------------------------}
  WHILE (i<>borne.fin) DO
  BEGIN
    j:=1;
    trouve:=FALSE;
    CASE pile[i].iconstr OF
    simvar,indvar: BEGIN
          nbr_arg:=0;
          WHILE (NOT trouve) AND (j<= LongReg) DO
          BEGIN
            IF reg[j].libre THEN
            BEGIN
              reg[j].val_reg:=value[pile[i].num_var];
              resultat:=reg[j].val_reg;
              reg[j].libre:=FALSE;
              trouve:=TRUE
            END
            ELSE j:=j+1
          END
        END;
    cons      : BEGIN
          nbr_arg:=0;
          WHILE (NOT trouve) AND (j<= LongReg) DO
          BEGIN
            IF reg[j].libre THEN
            BEGIN
              reg[j].val_reg:=pile[i].consval;
              resultat:=reg[j].val_reg;
              reg[j].libre:=FALSE;
              trouve:=TRUE
            END
            ELSE j:=j+1
          END
        END;
    unop,binop,fonction  : BEGIN
          nbr_arg:=pile[i].nbr_arg;
          WHILE (NOT trouve)AND(j<=LongReg)DO
          BEGIN
            IF reg[j].libre THEN
```

```
    BEGIN
      FOR k:=1 TO nbr_arg DO
        operande[k]:=reg[(j+k)-(nbr_arg+1)].val_reg;
      trouve:=TRUE;
      FOR k:=j-nbr_arg+1 TO j-1 DO
        reg[k].libre:=TRUE;
    END
    ELSE
      j:=j+1
END ;

CASE pile[i].num_oper OF
1:    resultat := operande[1] * operande[2];
2:    IF(operande[2]=0)
      THEN affichage(nl,DIV_0 )
      ELSE resultat := operande[1] DIV operande[2];
3:    resultat := operande[1] MOD operande[2];
4:    IF pile[i].iconstr=unop
      THEN resultat:=operande[1]
      ELSE resultat := operande[1] + operande[2];
5:    IF pile[i].iconstr=unop
      THEN resultat:=operande[1]
      ELSE  resultat := operande[1] - operande[2];
6:    IF(operande[1] = operande[2])
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
7:    IF(operande[1] <> operande[2])
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
8:    IF(operande[1] < operande[2])
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
9:    IF(operande[1] > operande[2])
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
10:   IF(operande[1] <= operande[2])
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
11:   IF(operande[1] >= operande[2])
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
12:   IF((operande[1]=1) AND (operande[2]=1))
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
13:   IF((operande[1]=1) OR (operande[2]=1))
      THEN resultat:=1{*vrai*}
      ELSE resultat:=0{*faux*};
14:   IF ( operande[1] = 0 )
      THEN resultat := 1  {*vrai*}
      ELSE resultat := 0  {*faux*};
15:   IF operande[1] >operande[2]
      THEN resultat:=operande[1]
      ELSE resultat:=operande[2];
16:   IF operande[1] <operande[2]
      THEN resultat:=operande[1]
      ELSE resultat:=operande[2];
17:   IF operande[1]=N
      THEN resultat:=N
```

```
                ELSE resultat:=0;
            END;
          END;
      END ;
      reg[j-nbr_arg].val_reg:=resultat;
      compteur:=compteur+1;

      i:=i+1;
    END;
    t:=resultat;
END;{*calcul*}


{*************************************************************************}
{************************ L'algorithme Bottom-Up ************************}
{*************************************************************************}

{-----------------------------------------------------------------
  Construction d'un tableau des chaines des fourchettes de calcul
  dans la pile des inéquations
-----------------------------------------------------------------}
PROCEDURE ParcourPile(pile:ty_tab_in );
  VAR
    num_eq,debut,i,j  :INTEGER;
    var_cour          :ty_var;
BEGIN
  num_eq:=0;
  par_g[1]:=pile[1].num_var;
  j:=1;
  debut:=1;
  FOR i:=2 TO (NbEl) DO
    IF (pile[i].num_var=0)  THEN
    BEGIN
      num_eq:=num_eq+1;
      new(chain_gen[j]);
      chain_gen[j]^.borne_eq.debut:=debut+1;
      chain_gen[j]^.borne_eq.fin:=i-1;
      chain_gen[j]^.chsuiv:=NIL;
      chain_gen[j]^.num_eq:=num_eq;
      debut:=i+1;
      IF (i<>NbEl) THEN
      BEGIN
        j:=j+1;
        par_g[j]:=pile[i+1].num_var;
      END;
    END;
  NbPar:=j;
END;{*ParcourPile*}

{-----------------------------------------------------------------
Construit le tableau de chaines, chaque élément de la chaîne
contient la suite de fourchettes
indiquant la position des inéquations d'intérêt dans la pile.
L'intérêt est de trouver des inéquations où la variable de numéro
 trouvé dans le tableau de chaînes figure comme opérande dans
le membre de droite.
-----------------------------------------------------------------}
PROCEDURE ConstrChain(pile:ty_tab_in;NbPar:INTEGER);
```

```
  VAR
    debut       :INTEGER;
    i,j,num_eq  :INTEGER;
    nouveau     :chain_var;
    precedent   :chain_var;
    tete        :chain_var;
    presence    :BOOLEAN;
    ch          :ty_chain;

BEGIN
{-------------------------------------------------------------------
  Initialisation
-------------------------------------------------------------------}
  FOR i:=1 TO NbVar DO
    ch[i]:=NIL;
{-------------------------------------------------------------------
  Corps
-------------------------------------------------------------------}
  FOR i:=1 TO NbPar DO
  BEGIN
    presence:=FALSE;
    tete:=NIL;
    nouveau:=NIL;
    debut:=1;
    num_eq:=0;
    FOR j:=2 TO NbEl DO BEGIN
{-------------------------------------------------------------------
  On cherche les variables égales à par_g[i] dans
  les parties droites des inéquations
-------------------------------------------------------------------}
      IF ((pile[j].iconstr=simvar) OR
        (pile[j].iconstr=indvar)) AND
        (par_g[i]=pile[j].num_var) AND
        (j<>debut)
        THEN presence:=TRUE;
      IF (pile[j].num_var=0) THEN
      BEGIN
        num_eq:=num_eq+1;
        IF (pile[j-1].num_oper IN [6,11]) AND
          presence
          THEN BEGIN
{-------------------------------------------------------------------
  Si l'inéquation de type x[i]>=f_i ou x[i]=f_i,
  alors, on construit un élément de la chaine
-------------------------------------------------------------------}
            precedent:=nouveau;
            new(nouveau);
            nouveau^.num_eq:=num_eq;
            nouveau^.borne_eq.debut:=debut+1;
            nouveau^.borne_eq.fin:=j-1;
            nouveau^.chsuiv:=NIL;
            IF tete=NIL THEN
              tete:=nouveau
            ELSE precedent^.chsuiv:=nouveau;
            presence:=FALSE;
          END;
        debut:=j+1;
      END;
```

```
      END;
      ch[i]:=tete;
    END;
    FOR i:=1 TO NbPar DO
      chain[par_g[i]]:=ch[i];
END;{*ConstrChain*}

{-----------------------------------------------------------------
  Calcule en deux étapes les points fixes d'un système
  d'inéquations
-----------------------------------------------------------------}
PROCEDURE AlgBotUp(pile:ty_tab_in;chain_gen:ty_chain;chain:ty_chain);
  VAR
    stack       : ty_stack;
    nouveau     : ty_stack;
    c           : chain_var;
    i,j,k,t     : INTEGER;
    link        : ARRAY[1..NbVar] OF BOOLEAN;
    num_eq      : INTEGER;
    borne       : ty_borne;

BEGIN
{-----------------------------------------------------------------
  Initialisation
-----------------------------------------------------------------}
  writeln(fich_out,' Résultat de la partie d''initialisation ') ;
  FOR i:=1 TO NbVarCour DO
  BEGIN
    value[i]:=0;
    link[i]:=FALSE;
  END;
  stack:=NIL;
{-----------------------------------------------------------------
  Première étape de l'algorithme
-----------------------------------------------------------------}
  FOR j:=1 TO NbPar DO
  BEGIN
    c:=chain_gen[j];
    borne:=c^.borne_eq;
    calcul(pile,borne,t);
    num_eq:=c^.num_eq;
    k:=par_g[num_eq];
    IF t> value[k] THEN BEGIN
{-----------------------------------------------------------------
  On empile la variable dans stack et on change la
  valeur courante de point fixe correspondant
-----------------------------------------------------------------}
      value[k]:=t;
      link[k]:=TRUE;
      new(nouveau);
      nouveau^.numvar:=k;
      nouveau^.varsuiv:=stack;
      stack:=nouveau;
    END;
  END;
  FOR i:=1 TO NbVarCour DO
  BEGIN
    write(fich_out,tab_var[i].vnom);
```

```
    IF tab_var[i].iconstr=tab THEN
      writeln(fich_out,tab_var[i].indice,':',value[i])
    ELSE writeln(fich_out,':',value[i])
  END;

{-----------------------------------------------------------------
  Deuxième étape
-----------------------------------------------------------------}
  writeln(fich_out,' Résultat de la partie de fermeture ') ;
  WHILE stack<>NIL DO BEGIN
{-----------------------------------------------------------------
  Dépiler le sommet de stack, trouver la chaine correspondante,
  effectuer le calcul
-----------------------------------------------------------------}
    i:=stack^.numvar;
    stack:=stack^.varsuiv;
    link[i]:=FALSE;
    c:=chain[i];
    WHILE c<> NIL DO
    BEGIN
      borne:=c^.borne_eq;
      calcul(pile,borne,t);
      num_eq:=c^.num_eq;
      c:=c^.chsuiv;
      j:=par_g[num_eq];

      IF (t>value[j]) THEN BEGIN
{-----------------------------------------------------------------
  On empile la variable dans stack et on change
  la valeur courante du point fixe correspondant
-----------------------------------------------------------------}
        value[j]:=t;
        IF NOT(link[j]) THEN
        BEGIN
          new(nouveau);
          nouveau^.numvar:=j;
          nouveau^.varsuiv:=stack;
          stack:=nouveau;
        END;
      END;
    END;
  END;
  FOR i:=1 TO NbVarCour DO
  BEGIN
    write(fich_out,tab_var[i].vnom);
    IF tab_var[i].iconstr=tab THEN
      writeln(fich_out,tab_var[i].indice,':',value[i])
    ELSE writeln(fich_out,':',value[i])
  END;

END{*AlgBotUp*};
PROCEDURE Alg1;
  VAR T1,T2,dT,Reste : Longint;
BEGIN
  compteur:=0;
  writeln(fich_out,'    Algorithme Bottom-Up');
  ParcourPile(pile);
  ConstrChain(pile,NbPar);
```

```
  GetTime(h,m,s,c_s);
  AlgBotUp(pile,chain_gen,chain);
  GetTime(heure,minute,seconde,centiem_sec);
{-------------------------------------------------------------------
  Calcul du temps dépensé par l'algorithme Bottom-Up pour
  calculer le point fixe
-------------------------------------------------------------------}
  T1:=((h*60+m)*60+s)*100+c_s;
  T2:=((heure*60+minute)*60+seconde)*100+centiem_sec;
  dT := T2 - T1;
  H1 := dT DIV 360000;
  Reste := dT - H1*360000;
  M1 := Reste DIV 6000;
  Reste := Reste - M1*6000;
  S1 := Reste DIV 100;
  C_S1 := Reste - S1*100;

  writeln(fich_out,'Temps d''exécution par l''algorithme Bottom-Up : ',h1,':',m1,':',s1,':',c_s1);
  writeln(fich_out,'Nombre d''opérations effectuées par l''algorithme Bottom-Up : ',compteur);
  compt_alg1:=compteur;
END;


{*************************************************************************}
{**********************L'algorithme Top-Down*********************************}
{*************************************************************************}

{-------------------------------------------------------------------
  Cherche le numéro d'inéquation
  correspondant à la variable en question
-------------------------------------------------------------------}
FUNCTION NumPar(num:INTEGER):INTEGER;
  VAR
    i           : INTEGER;
    presence    : BOOLEAN;
BEGIN
  presence:=FALSE;
  i:=1;
  WHILE (i<= NbPar) AND (NOT presence) DO
  BEGIN
    IF par_g[i]=num THEN
    BEGIN
      presence:=TRUE;
      NumPar:=i;
    END
    ELSE i:=i+1
  END
END{NumPar};
{-------------------------------------------------------------------
  Construit une suite de dépendance statique
-------------------------------------------------------------------}
PROCEDURE ConstrSuit(pile:ty_tab_in;chain_gen:ty_chain);
  VAR
    i,j                                 : INTEGER;
    nouveau,precedent,el_suite          : el_dg;
    num                                 : INTEGER;
    var_aux                             : el_dg;
    absence                             : BOOLEAN;
```

```
    ch                                          : ty_dg;
BEGIN
{-------------------------------------------------------------------
  Initialisation
-------------------------------------------------------------------}
  j:=1;
  ch[1]:=NIL;
{-------------------------------------------------------------------
  Corps
-------------------------------------------------------------------}
  FOR i:=1 TO NbEl DO
  BEGIN
    IF pile[i].num_var =0 THEN BEGIN
      j:=j+1;
      ch[j]:=NIL;
    END
    ELSE
      IF (pile[i].iconstr=simvar) OR (pile[i].iconstr=indvar)   THEN
      BEGIN
        el_suite:=ch[j];
        absence:=TRUE;
        WHILE absence AND (el_suite<>NIL) DO
{-------------------------------------------------------------------
  Teste si un élément est déjà présent dans la table
-------------------------------------------------------------------}
          IF pile[i].num_var=el_suite^.num_var THEN
            absence:=FALSE
          ELSE
            el_suite:=el_suite^.el_suiv;
        IF absence THEN BEGIN
{-------------------------------------------------------------------
  Construction d'un élément d'une suite
-------------------------------------------------------------------}
          precedent:=nouveau;
          new(nouveau);
          nouveau^.num_var:=pile[i].num_var;
          nouveau^.el_suiv:=NIL;
          IF ch[j]=NIL THEN
            ch[j]:=nouveau
          ELSE
            precedent^.el_suiv:=nouveau
        END
      END
  END;
  FOR i:=1 TO NbPar DO
    suite[par_g[i]]:=ch[i];
END{ConstrSuit};
{-------------------------------------------------------------------
  Annule le codomaine d'un graphe de dépendance pour le domaine
  égal à x, organise la boucle pour annuler les codomaines
  des domaines correspondants aux codomaines déjà annullées
-------------------------------------------------------------------}
PROCEDURE MettreNul(x:INTEGER;VAR  dg:ty_dg);
  VAR
    precedent,suivant,tete      :el_dg;
    pour_libere,aux1,aux2       :el_dg;

BEGIN
```

```
  pour_libere:=dg[x];
  optimal[x]:=0;
  dg[x]:=NIL;
  tete:=NIL;
  WHILE pour_libere<> NIL DO
  BEGIN
    IF (optimal[pour_libere^.num_var]=1)  THEN
    BEGIN
      aux1:=dg[pour_libere^.num_var];
{-------------------------------------------------------------------
  Annulation d'un élément de graphe spécifié
-------------------------------------------------------------------}
      dg[pour_libere^.num_var]:=NIL;
      optimal[pour_libere^.num_var]:=0;
{-------------------------------------------------------------------
  Fin d'annulation
-------------------------------------------------------------------}
      suivant:=aux1;
      WHILE suivant<>NIL DO
      BEGIN
        precedent:=suivant;
        suivant:=suivant^.el_suiv
      END;
{-------------------------------------------------------------------
  La chaine des éléments à annuler est placée en tête
  d'une chaine à libérer
-------------------------------------------------------------------}
      IF tete<>NIL THEN
        precedent^.el_suiv:=tete;
      tete:=aux1
    END;
    aux2:=pour_libere^.el_suiv;
{-------------------------------------------------------------------
  On libère les cellules occupées par des éléments déjà annulés
-------------------------------------------------------------------}
    dispose(pour_libere);
    IF aux2<> NIL
    THEN pour_libere:=aux2
    ELSE
{-------------------------------------------------------------------
  On constate la fin d'une chaîne à libérer,
  on revient à la tête et on continue à libérer
  si la tête n'est pas vide
-------------------------------------------------------------------}
    BEGIN
      pour_libere:=tete;
      tete:=NIL
    END
  END;
END{*MettreNull*};
{-------------------------------------------------------------------
  Ajoute un élément x dans le codomaine de graphe de dépendance,
  le domaine valant y
-------------------------------------------------------------------}
PROCEDURE CreerDg(y:INTEGER;x:INTEGER;VAR dg:ty_dg);
  VAR
    absence     : BOOLEAN;
    elem        : el_dg;
```

```
BEGIN
  elem:=dg[y];
  absence:=TRUE;
  WHILE absence AND (elem<>NIL) DO
    IF x=elem^.num_var THEN
      absence:=FALSE
    ELSE
      elem:=elem^.el_suiv;
  IF absence THEN BEGIN
{-------------------------------------------------------------------
  On crée un nouvel élement de codomaine du graphe de dépendance
-------------------------------------------------------------------}
    precedent[y]:=nouveau[y];
    new(nouveau[y]);
    nouveau[y]^.num_var:=x;
    nouveau[y]^.el_suiv:=NIL;
    IF dg[y]=NIL
      THEN
      dg[y]:=nouveau[y]
    ELSE
      precedent[y]^.el_suiv:=nouveau[y];
  END
END{*CreerDg*};

{-------------------------------------------------------------------
  Evalue la transformation tau, trouve la valeur du point fixe de
  transformation pour la variable numéro x
-------------------------------------------------------------------}
PROCEDURE eval(x:INTEGER;VAR dg:ty_dg);
  VAR
    resultat,num,y    : INTEGER;
    domcodom          : el_dg;
    borne             : ty_borne;

BEGIN
  IF (NOT ics[x]) AND (optimal[x]=0) THEN BEGIN
{-------------------------------------------------------------------
  lancement du calcul
  si le calcul d'un point fixe pour la variable numéro x n'est pas
  encore initialisé, et que le valeur n'est pas optimale
-------------------------------------------------------------------}
    ics[x]:=TRUE;
    domcodom:=suite[x]^.el_suiv;
    IF pt[x].valeur=-1 THEN
    BEGIN
      pt[x].valeur:=bottom;
      value[x]:=bottom
    END;

    REPEAT
      optimal[x]:=1;
      IF domcodom<> NIL THEN
      BEGIN
        y:=domcodom^.num_var;
        eval(y,dg);
        CreerDg(y,x,dg);
        domcodom:=domcodom^.el_suiv;
        WHILE domcodom<>NIL DO
```

```
          BEGIN
            y:=domcodom^.num_var;
            eval(y,dg);
            CreerDg(y,x,dg);
            domcodom:=domcodom^.el_suiv
          END;
        END;
        num:=NumPar(x);
        borne:=chain_gen[num]^.borne_eq;
        calcul(pile,borne,resultat);
        IF resultat> pt[x].valeur THEN
        BEGIN
          pt[x].valeur:=resultat;
          value[x]:=resultat;
          IF dg[x]<>NIL THEN
            MettreNul(x,dg);
          IF optimal[x]=0 THEN
          BEGIN
            ics[x]:=FALSE;
            eval(x,dg)
          END
        END
      UNTIL optimal[x]=1;
      ics[x]:=FALSE
    END
END{*eval*};

PROCEDURE ComputeFixpoint(x: INTEGER; dg:ty_dg; VAR pt: ty_pt);
  VAR i:INTEGER;
BEGIN
{-------------------------------------------------------------------
  Initialisation
-------------------------------------------------------------------}
  FOR i:=1 TO NbVarCour DO

  BEGIN
    pt[i].valeur:=-1;
    ics[i]:=FALSE;
    optimal[i]:=0;
    dg[i]:=NIL;
    precedent[i]:=NIL;
    nouveau[i]:=NIL;
  END;
  ConstrSuit(pile,chain_gen);
  eval(x,dg)
END{*ComputeFixpoint*};
{-------------------------------------------------------------------
  Organise l'interface Homme-Machine pour introduire le nombre
  de calculs à effectuer pour l'algorithme local de calcul de
  point fixe, et renvoyer à l'utilisateur le résultat des calculs.
  Appelle la procédure centrale de l'algorithme
-------------------------------------------------------------------}
PROCEDURE Alg2;
  VAR
    x                : INTEGER;
    i                : INTEGER;
    T1,T2,dT,Reste   : Longint;
    dH, dM, dS, dC   : Word;
```

```
BEGIN
  compteur:=0;

  writeln(fich_out,'      Algorithme de B. LeCharlier');

  FOR i:=1 TO NbVarCour DO
  BEGIN
    write(i,': ',tab_var[i].vnom);
    IF tab_var[i].iconstr=tab THEN
      writeln(tab_var[i].indice)
    ELSE writeln
  END;

  writeln('Vous pouvez choisir une variable pour calculer le point fixe.');
  writeln('Tapez le numéro de la variable choisie : ');
  read(x);
  writeln(fich_out,' L''algorithme Top-Down calcul le point fixe '
  'pour la variable X',x);

  GetTime(h,m,s,c_s);

  ComputeFixpoint(x,dg,pt);

  writeln(fich_out,' Résultat des calculs par l''algorithme Top-Down :');
  FOR i:=1 TO NbVarCour DO
  BEGIN
    write(fich_out,tab_var[i].vnom);
    IF tab_var[i].iconstr=tab THEN
      writeln(fich_out,tab_var[i].indice,':',pt[i].valeur)
    ELSE writeln(fich_out,':',pt[i].valeur)
  END;

  GetTime(heure,minute,seconde,centiem_sec);
{-----------------------------------------------------------------
  Calcul du temps dépensé par l'algorithme Bottom-Up pour
  calculer le point fixe
-----------------------------------------------------------------}
  T1:=((h*60+m)*60+s)*100+c_s;
  T2:=((heure*60+minute)*60+seconde)*100+centiem_sec;
  dT := T2 - T1;
  H := dT DIV 360000;
  Reste := dT - H*360000;
  M := Reste DIV 6000;
  Reste := Reste - M*6000;
  S := Reste DIV 100;
  C_S := Reste - S*100;

  writeln(fich_out,'Nombre d''opérations effectuées par l''algorithme '+
  'de B. LE CHARLIER : ',compteur,'.');
  writeln(fich_out,'Pour comparaison, le nombre d''opérations pour '+
  'l''algorithme');
  writeln(fich_out,'Bottom-Up Útait de : ',compt_alg1,'.');
  writeln;
  writeln(fich_out,'Temps d''exécution de l''algorithme de B.LE CHARLIER : ',h,':',m,':',s,':',c_S,'.');
  writeln(fich_out,'Temps d''exécution de l''algorithme Bottom-Up :      ',h1,':',m1,':',s1,':',c_S1,'.');
END;{*Alg2*}

END.{*unit*}
```

fichier SystIneq.pas (programme principal)

```
PROGRAM SystIneq;

{$M 30000,0,655360}

USES DefGlob, BILire, BiblFonc,AnSynt,CrPile,Algorit;

VAR
  prog                                  :ty_constr;
  fnom_out,fnom,fnom1,fnom_complet      :STRING[30];
  num_prog                              :INTEGER;

procedure CreerFich(N:integer;num_prog:integer;var fich_en:text)   ;forward;

{-------------------------------------------------------------------
  Crée des fichiers de texte contenant les exemples de systèmes
  d'inéquations. Ces exemples sont proposés pour l'étude expérimentale
  des deux algorithmes Top-Down et Bottom-Up.
  Les noms des fichiers de texte fournis par l'auteur sont:
  "boucle1", "boucle2", "boucle3", "boucle4", et "boucle5".
  Il reste à l'utilisateur à taper le nom correspondant à son choix.
-------------------------------------------------------------------}
PROCEDURE CreerFich(N:INTEGER;num_prog:INTEGER;VAR fich_en:TEXT);

  VAR
    i,j  :INTEGER;
BEGIN
  IF num_prog=1 THEN
{-------------------------------------------------------------------
  Création du premier exemple, de nom "boucle1"
-------------------------------------------------------------------}
  BEGIN
    writeln(fich_en,'program boucle1;');
    writeln(fich_en,'  var');
    writeln(fich_en,'   x:array[1..',N,'] of integer;');
    writeln(fich_en,'  begin');
    FOR i:=2 TO N DO
      writeln(fich_en,'    x[',i,']=min(',N,',x[',i-1,']+1);');
    writeln(fich_en,'    x[1]=1;');
    writeln(fich_en,'  end.');
    writeln(fich_en,' ?')
  END

  ELSE
    IF num_prog=2 THEN
{-------------------------------------------------------------------
  Création du deuxième exemple, de nom "boucle2"
-------------------------------------------------------------------}
      BEGIN
        writeln(fich_en,'program boucle2;');
        writeln(fich_en,'  var');
        writeln(fich_en,'   x:array[1..',N,'] of integer;');
        writeln(fich_en,'  begin');
        FOR i:=1 TO N-3 DO
```

```
        writeln(fich_en,'    x[',i,']=max(x[',i+1,'],x[',N-2,']);');
      writeln(fich_en,'    x[',N-2,']=comp(x[',N-1,']);');
      writeln(fich_en,'    x[',N-1,']=min(',N,',x[',N,']+1);');
      writeln(fich_en,'    x[',N,']=max(x[1],x[',N-1,']);');
      writeln(fich_en,' end.');
      writeln(fich_en,' ?')
    END

    ELSE
      IF num_prog=3 THEN
{-------------------------------------------------------------------------
  Création du troisième exemple, de nom "boucle3"
-------------------------------------------------------------------------}
      BEGIN
        writeln(fich_en,'program boucle3;');
        writeln(fich_en,' var');
        writeln(fich_en,'  x:array[1..',N,'] of integer;');
        writeln(fich_en,' begin');
        writeln(fich_en,'    x[1]=1;');
        FOR i:=N DOWNTO 2 DO
          writeln(fich_en,'    x[',i,']=min(',N,',x[',i-1,']+1);');

        writeln(fich_en,' end.');
        writeln(fich_en,' ?')
      END
      ELSE
        IF num_prog=4 THEN
{-------------------------------------------------------------------------
  Création du quatrième exemple, de nom "boucle4"
-------------------------------------------------------------------------}
        BEGIN
          writeln(fich_en,'program boucle4;');
          writeln(fich_en,' var');
          writeln(fich_en,'  x:array[1..',N,'] of integer;');
          writeln(fich_en,' begin');
          j:=0;
          WHILE j<11 DO
          BEGIN
            FOR i:=1 TO N DO
            BEGIN
              IF (i MOD 10=j) AND (i<>N-2) AND (i<>N-1) AND (i<>N)
                THEN  writeln(fich_en,'    x[',i,']=max(x[',i+1,'],x[',N-2,']);')
              ELSE
                IF (i MOD 10=j) AND (i=N-2)
                  THEN writeln(fich_en,'    x[',N-2,']=comp(x[',N-1,']);')
                ELSE
                  IF (i MOD 10=j) AND (i=N-1)
                    THEN writeln(fich_en,'    x[',N-1,']=min(',N,',x[',N,']+1);')
                  ELSE
                    IF (i MOD 10=j) AND (i=N)
                      THEN writeln(fich_en,'    x[',N,']=max(x[1],x[',N-1,']);');
            END;
            j:=j+1
          END;
          writeln(fich_en,' end.');
          writeln(fich_en,' ?')
        END
        ELSE
```

```
            IF num_prog=5 THEN
{-------------------------------------------------------------------------
  Création du cinquième exemple, de nom "boucle5"
-------------------------------------------------------------------------}
            BEGIN
              writeln(fich_en,'program boucle5;');
              writeln(fich_en,'  var');
              writeln(fich_en,'   x:array[1..',N,'] of integer;');
              writeln(fich_en,'  begin');
              j:=0;
              WHILE j<11 DO
              BEGIN
                FOR i:=1 TO N DO
                BEGIN
                  IF (i MOD 10=j) AND (i<>1)
                    THEN writeln(fich_en,'    x[',i,']=min(',N,',x[',i-1,']+1);')
                  ELSE
                    IF (i MOD 10=j) AND (i=1)
                      THEN  writeln(fich_en,'    x[1]=1;');
                END;
                j:=j+1
              END;
              writeln(fich_en,'  end.');
              writeln(fich_en,' ?')
            END;
  close(fich_en);
END {CreerFich};

BEGIN{* Programme principal *}
write('Introduisez le nom du fichier d"inéquations: ' );
{-------------------------------------------------------------------------
  L'utilisateur a le choix: soit, il crée le fichier de texte
  contenant un système d'inéquations; soit, il tape le nom
  d'un exemple proposé par l'auteur
-------------------------------------------------------------------------}
readln(fnom);
fnom_complet:='\users\curr\'+fnom+'.txt';
assign(fich_en,fnom_complet);

{-------------------------------------------------------------------------
  L'utilisateur est invité à introduire le nom du fichier qui
  contiendra le résultat des calculs
-------------------------------------------------------------------------}
write('Introduisez le nom du fichier de résultats: ');
readln(fnom1);
fnom_out:='\users\curr\'+fnom1+'.txt';
assign(fich_out,fnom_out);
rewrite(fich_out);
writeln(fich_out,'Le système d"inéquations à traiter porte le nom:',fnom);

IF (fnom='boucle1') OR (fnom='boucle2') OR (fnom='boucle3') OR (fnom='boucle4')
  OR (fnom='boucle5')
  THEN
BEGIN
  writeln('Donnez la valeur souhaitée pour N.');
  write('N=');
  readln(N);
  writeln(fich_out,'Le nombre de variables dans notre système d"inéquations est ',N);
```

```
  IF fnom='boucle1' THEN
    num_prog:=1
  ELSE
    IF fnom='boucle2' THEN
      num_prog:=2
    ELSE
      IF fnom='boucle3' THEN
        num_prog:=3
      ELSE
        IF fnom='boucle4' THEN
          num_prog:=4
        ELSE
          IF fnom='boucle5' THEN
            num_prog:=5;
  rewrite(fich_en) ;
  CreerFich(N,num_prog,fich_en);
END;

reset(fich_en);
{-------------------------------------------------------------------
  Construction de la représentation interne à partir d'une
  représentation externe des inéquations
-------------------------------------------------------------------}
writeln('Début de construction de la représentation interne');
prog:=sxprog;
writeln('Fin de construction de la représentation interne');
writeln(fich_out,'La pile du système d''inéquations est la suivante:');
{-------------------------------------------------------------------
  Construction de la pile à partir de la représentation interne
-------------------------------------------------------------------}
ConstrPileIneq(prog);


{*****************************************************************}
{**********************Algorithme Bottom-Up***********************}
{*****************************************************************}

Alg1;

writeln('Fin de l''algorithme Bottom_Up.');
writeln('Appuyer sur la touche "ENTER" pour continuer...');
readln;


{*****************************************************************}
{**********************Algorithme Top-Down************************}
{*****************************************************************}

Alg2;

close(fich_en);
close(fich_out);
END.
```

Annexe 3. PROCEDURES ET FONCTIONS DU PROGRAMME

1. Analyseur lexical

Procédure liresymb

Le bloc principal se compose de :
- la formation de la chaîne l_chaîne
- l'analyse de la chaîne formée l_chaîne.

Formation de la chaîne

Appelle: ♥ la procédure *putchar*
♥ la fonction *getchar*
Paramètre: ◊
Présentation:

- lire, grâce à getchar, tant que le caractère s est égal à un espace ou au retour chariot

- si s est un séparateur
alors traitement associé aux séparateurs présents dans un programme SYSTINEQ
sinon traitement_chaîne

◊ traitement associé aux séparateurs

placer le séparateur dans l_chaîne
si le séparateur est l'apostrophe
alors traitement_constante_forme_'c'
où c est un des caractères énumérés dans
l'ensemble des valeurs de type caractère

◊ **traitement_constant _forme_'c'**

lire le caractère s suivant
concaténer le caractère s avec l_chaîne
lire le caractère s suivant
concaténer le caractère s avec l_chaîne

◊ **traitement_chaîne**
{formation de la chaîne l_chaîne}

répéter la concaténation du caractère s et de l_chaîne
lecture du caractère suivant s
tant que le caractère s n'est pas un séparateur
{ ce caractère s lu ne doit pas être perdu }
garder en mémoire grâce à *putchar*

DESCRIPTION DE LA FONCTION GETCHAR

Nom: ♦ getchar;
Type ♠ char;
Argument: ◊
Présentation:

si un caractère déjà lu est présent en mémoire
(tester grâce au booléen caract_présent)
alors le caractère lu caract est celui présent en mémoire caract_mem
{il n'y aura plus de caractère présent en mémoire pour un prochain appel}
sinon on lit dans le fichier d'entrée un caractère que l'on place dans caract
le caractère caract est mis en majuscule
le résultat est assigné à *getchar*

DESCRIPTION DE LA PROCÉDURE PUTCHAR

Nom: ♦ putchar;
Paramètre: ◊ s de type char;
Effet: • place en mémoire le caractère s dans caract_mémoire;
• affecte à un booléen caract_présent la valeur true .

Analyse de la chaîne

si l_chaîne est un booléen
alors traitement _booleen
{ consiste en l'affectation à symb de la représentation interne
du symbole de base de genre booléen et d'identité précise }
sinon si l_chaîne est un Symbspec (testé par la fonction *InSymbspes*)
alors traitement_symbspec
{ qui consiste en l'affectation à symb de la représentation interne du
symbole de base de genre specsymb et d'identité précise }
sinon si l_chaîne commence par un coté
alors
si l_chaîne n'a pas un coté en position 3
alors traitement_errsymb
{ qui consiste en l'affectation à symb de la représentation interne du
symbole de base de genre errsymb }
sinon traitement_caractère
{ qui consiste en l'affectation à symb de la représentation interne du
symbole de base de genre caractère et d'identité calculé grâce à la
fonction NumTab*OrdTabcar* }
sinon si l_chaîne est '?'
alors traitement_nosymb
{ qui consiste en l'affectation à symb de la représentation interne du
symbole de base (de genre nosymb) }
sinon si l_chaîne commence par une lettre
alors traitement_identificateur
{ qui consiste en l'affectation à symb de la représentation interne du
symbole de base (de genre identificateur et d'identité précise) }

sinon si l_chaîne commence par un chiffre
alors traitement_chiffre
{ qui consiste en l'affectation à symb de la représentation interne du symbole de base (de genre entier et d'identité précise) }
sinon traitement_errsymb
{ qui consiste en l'affectation à symb de la représentation interne du symbole de base (de genre errsymb) }

DESCRIPTION DE LA FONCTION InSymbspec

Nom: ♦ InSymbspec;
Type: ♠ boolean;
Argument: ◊ s de type chaîne;
Présentation:

si présence de s dans Symbspec
alors TRUE est assigné à InSymbspec
sinon FALSE est assigné à *InSymbspec*

DESCRIPTION DE LA FONCTION NumTabcar

Nom: ♦ NumTabcar;
Type: ♠ integer;
Argument: ◊ c de type char;
Effet: • assigne à *NumTabcar* le rang i de c, dans l'énumération formée par l'ensemble des valeurs de type caractère.

2. Analyseur syntaxique

Fonction sxprog

Nom : ♦ sxprog : ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne du programme SYSINEQ analysé, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte;

Implémentation :

liresymb : lecture d'un symbole
si pas 'PROGRAM'
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
sinon valeur d'indicateur : prog pour iconstr
liresymb : lecture d'un symbole
si pas identificateur
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
valeur d'indicateur : cval pour pnom
liresymb : lecture d'un symbole
si pas ';'
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
liresymb : lecture d'un symbole
si pas ('VAR' ou 'BEGIN')
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
si pas 'VAR'
alors valeur indicateur : nil pour lvar
sinon *liresymb* : lecture d'un symbole
répéter appel à *sxvar* pour le traitement des déclarations de variables
avec la construction de lvar
{ sortie de sxvar avec la lecture de ; }
si pas ';'
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
liresymb : lecture d'un symbole
si pas ('BEGIN')
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
tant que le symbole est un identificateur
traitement du bloc principal en appelant *sxineq*
si pas '.' { après le end final }
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
liresymb : lecture d'un symbole
si pas '?'
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
assigner à *sxprog* le ptr_prg

Fonction sxfacteur

Nom: ♦ sxfacteur ;
Type: ♠ ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne du facteur analysé, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte;

Implémentation :

s'il existe un préfixe facteur-maximal de la suite de symboles à traiter,
càd si l'on trouve (a) une constante
ou (b) une expression
ou (c) un appel de fonction
ou (d) une expression parenthétisée
alors on en construit la représentation interne :
pour (a), d'une constante;
pour (b), on appelle la fonction de sxexpr;
pour (c), on construit la représentation interne de la fonction,
pour les arguments de la fonction, on fait appel à sxexpr;
pour (d), il suffit d'ignorer les parenthèses et de rappeler sxfacteur (appel récursif)
dans tous les cas, la fonction renvoie un pointeur
vers la représentation interne construite

sinon terminaison immédiate avec message d'erreur affiché à l'écran

Fonction sxterme

Nom: ♦ sxterme;
Type: ♠ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne du terme analysé, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte

Implémentation :

s'il existe un préfixe terme-maximal de la suite de symboles à traiter
alors appel à *sxfacteur*
test d'une extension possible du terme
extension ?
OUI → appel à *sxfacteur* et traitement de l'opérateur multiplicatif
NON → -
le résultat de la fonction est le pointeur vers la représentation interne du terme
sinon erreur syntaxique déclarée et brusque arrêt de l'analyse syntaxique

Fonction sxaexpr

Nom: ♦ sxaexpr ;
Type: ♠ ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne de l'expression arithmétique analysée, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte;

Implémentation :

présence de errsymb ou nosymb dans le premier symbole de la suite de symboles à traiter ?
OUI → arrêt pour cause d'erreur
NON → construction de la représentation interne après un appel à *sxterme*
test de la présence d'un sign '-'
sign '-' présent ?
NON → appel à *sxterme*
test d'une extension possible de l'expression arithmétique ainsi découverte
opérateur multiplicatif présent ?
OUI → traitement de cet opérateur et appel à *sxterme*
NON → -
le résultat de la fonction est le pointeur vers la représentation interne

Fonction sxneg

Nom: ♦ sxneg ;
Type: ♠ ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne de la négation analysée, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte;

Implémentation :

présence de errsymb ou nosymb dans le premier symbole de la suite de symboles à traiter ?
OUI → arrêt pour cause d'erreur
NON →recherche de la présence d'un "NOT" au début de la suite de symboles à traiter
premier symbole = "NOT" ?
OUI → on est prévenu qu'on est en présence d'un opérateur unaire,
celui-ci étant la négation
NON → on sait qu'on aura affaire à un opérateur binaire ou à une simple expression arithmétique
appel à *sxaexpr*
test d'une extension possible de la négation
présence d'un opérateur relationnel ?
OUI → **si** on est dans le cas d'une négation
alors la négation est de la forme "not aexpr1 w aexpr2"
sinon la négation est de la forme "not aexpr"
NON → **si** on est dans le cas d'une négation
alors la négation est de la forme "not aexpr"
sinon elle est de la forme "aexpr"
le résultat de la fonction est le pointeur sur la construction engendrée,
càd sur la représentation interne de la négation analysée

Fonction sxconj

Nom: ♦ sxconj ;
Type: ♠ ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne de la conjonction analysée, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte

Implémentation :

erreur lors de la lecture du premier symbole de la suite de symboles à traiter ?
OUI → arrêt de l'analyse syntaxique avec impression d'un message d'erreur
le plus clair possible
NON → appel à *sxneg*
test d'une extension possible de la conjonction
présence de symbole 'AND'?
OUI → construction de la représentation interne de cet opérateur binaire
avec appel récursif à *sxconj*
NON → -
le résultat de la fonction est le pointeur sur la construction engendrée,
càd sur la représentation interne de la conjonction analysée

Fonction sxexpr

Nom: ♦ sxexpr;
Type: ♠ ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne de l'expression analysée, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte;

Implémentation :

erreur lors de la lecture du premier symbole de la suite de symboles à traiter ?
OUI → arrêt et message d'erreur
NON → appel à *sxconj*
test de la présence d'un or
présence du symbole 'OR' ?
OUI → construction de la représentation interne
de cet opérateur binaire
avec l'aide d'un appel récursif à *sxexpr*
NON → -
le résultat de la fonction est le pointeur vers la construction interne générée

Fonction sxineq

Nom: ♦ sxineq ;
Type: ♠ ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne de l'inéquation analysée, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte;
Implémentation :

s'il existe un préfixe ineq-maximal de la suite de symboles à traiter,
càd si on trouve
(a) une inéquation
ou (b) un système d'inéquations
alors on traite ce préfixe en construisant la représentation interne de ces inéquations (suivant la description qui en a été faite précédemment)
pour (a), on construit la représentation interne d'une telle inéquation,
à l'aide d'un pointeur sur une expression (appel à *sxexpr*)
pour (b), on construit une liste chaînée de pointeurs sur des représentations internes d'expressions en faisant de multiples appels récursifs (autant qu'il y a d'inéquations) à *sxineq*
dans tous les cas, la fonction retourne un pointeur sur la représentation interne construite
sinon l'appel de *sxineq* se termine anormalement
arrêt immédiat après impression d'un message indiquant l'erreur probable

Fonction sxvar

Nom: ♦ sxvar;
Type: ♠ty_constr;
Paramètres: ◊
Effet: • la fonction renvoie un pointeur vers la représentation interne des déclarations de variables analysées, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte
Présentation:

```
si pas identificateur
alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        répéter { construction de tête qui est la suite d'identificateurs }
                { tête sera assignée à lvnom ou ltnom dans la suite }
                liresymb : lecture d'un symbole
                        {lecture de , ou : après identificateur }
                si ','
                alors liresymb : lecture d'un symbole
                si pas identificateur
                alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        tant que le symbole n'est pas ':'
        liresymb : lecture d'un symbole
        si pas ( 'CHAR' ou 'BOOLEAN' ou 'INTEGER' ou 'ARRAY' )
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        si ( 'CHAR' ou 'BOOLEAN' ou 'INTEGER' )
        alors valeur indicateur : vrai pour iconstr
        si 'CHAR'
        alors valeur indicateur : caractère pour vartype
         si 'BOOLEAN'
        alors valeur indicateur : booleen pour vartype
                si 'INTEGER'
        alors valeur indicateur : entier pour vartype
                assigner à lvnom tête construit avant
                        liresymb : lecture d'un symbole
                        assigner à sxvar le p_constr
        sinon appel à la procédure tab_var
```

Fonction tab_var

Nom: ♦ tab_var ;
Paramètres: ◊
Effet: • la procédure renvoie un pointeur vers la représentation interne des déclarations de variables de type tableau analysées, ou bien s'arrête en cours d'exécution (après impression d'un message d'erreur) si la construction n'est pas syntaxiquement correcte;

Présentation:

```
valeur indicateur : tab pour iconstr
liresymb : lecture d'un symbole
        si pas '['
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        liresymb : lecture d'un symbole
        si pas entier
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        valeur indicateur : consval pour db
        liresymb : lecture d'un symbole
        si pas '.'
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        liresymb : lecture d'un symbole
        si pas '.'
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        liresymb : lecture d'un symbole
        si pas entier
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        valeur identificateur : consval pour fb
        liresymb : lecture d'un symbole
        si pas ']'
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        liresymb : lecture d'un symbole
        si pas 'OF'
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        liresymb : lecture d'un symbole
        si pas ( 'CHAR' ou 'BOOLEAN' ou 'INTEGER' )
        alors traitement d'erreur par l'impression d'un message d'erreur adéquat
        sinon
        si 'CHAR'
        alors valeur indicateur : caractère pour vartype
        si 'BOOLEAN'
        alors valeur indicateur : booléen pour vartype
        si 'INTEGER'
        alors valeur indicateur : entier pour vartype
        assigner à lvnom tete construit avant
        liresymb : lecture d'un symbole
        assigner à sxvar le p_constr
```

Procédure ConstrTabVar

Nom: ♦ ConstrTabVar;

Paramètres: ◊ x de type ty_constr;

Effet: • construit la table des variables à partir de la représentation interne des déclarations de variables,
le type d'un élément de la table des variables est défini comme suit :

```
ty_var= record
        iconstr:tconstr;
        vnom:chaîne;
        vtype:valtype;
        indice:integer
end;
```

• renvoie le nombre des variables courantes.

Présentation:

```
assigner le pointeur vers la représentation interne de déclaration des variables à dec_var
{on prend la partie de déclaration des variables de notre illustration}

tant que dec_var pas nil
        choix entre correspondant à le champ iconstr de variable dec_var
        valeur indicateur "tab" :
                assigner le pointeur vers la répresentation interne
                de première suite de variables de type tableau à aux1
                tant que la suite des noms des variables avec iconstr=tab,
                        la borne inférieure et la borne supérieure sont detérminées
                        et le type fixé n'est pas vide
                        à partir de valeur de la borne inférieure
                        jusqu'à la valeur de la borne supérieure
                                construire un élément de la table de variables
                                avancer d'un élément dans la table des variables
                avancer d'une variable dans la suite des noms des variables
        valeur indicateur "vrai" :
                assigner le pointeur vers la représentation interne
                de la première suite variables de type tableau à aux2
                tant que la suite des noms des variables avec iconstr=vrai
                et le type fixé n'est pas vide
                        construire un élément de la table des variables
                        avancer d'une variable dans la suite des noms des variables
                        avancer d'une déclaration dans la suite des déclarations des variables
                        assigner le nombre des variables construites dans la table des variables
                                à NbVarCour
```

3. Constructeur de la pile des inéquations

Procédure ConstrPileIneq

Nom: ♦ ConstrPileIneq;
Paramètres: ◊ prg de type ty_constr;
Effet: • renvoie la pile des inéquations composées des opérateurs et des opérandes d'un système d'inéquations; chaque inéquation dans la pile est séparée par le caractére '! ',

le type d'un élément de la table des inéquations est:

```
elem_tab_ineq=record
        valeur  :integer;
        case iconstr : tconstr of
        simvar,indvar :  (num_var :     integer;
                          vartype :     valtype;
                          ind :         integer);
        unop,binop,fonction:
                          ( num_oper :  integer;
                          optype : valtype;
                          nbr_arg :     integer);
        cons :            ( constype :  valtype;
                          consval :     integer);
end;
```

Présentation:

appel à la procédure *ConstrTabVar*
appel à la procédure *ConstrTabFonc*
assigner le pointeur vers la représentation interne des déclarations de variables à var_ineq
tant que var_ineq pas nil
    appel à la procédure PileIneq {construit un élément de pile pour une inéquation }
    mettre le séparateur d'inéquations {le caractère '!' }
    assigner la position d'un segment de la pile pour l'inéquation suivante à n
    passer à la présentation de l'inéquation suivante

Procédure PileIneq

Nom: ♦ PileIneq;

Paramètres: ◊ n de type integer {représente le début d'un segment de pile correspondant à une inéquation} ;
◊ x de type ty_constr;

Effet: • renvoie la partie de la pile des inéquations composées des opérateurs et des opérandes d'une inéquation;
• renvoie la valeur num indiquant la position de fin de l'inéquation dans la pile;

Présentation:

```
choix entre correspondant à la valeur de la champ iconstr de variable x
valeur est " fonction":
        assigner le pointeur vers la représentation interne des arguments de la fonction à y
        tant que y n'est pas nil
                appel récursif de la procédure PileIneq
                avancer d'une unité dans la représentation interne d'une inéquation
                appel de la fonction ConsultTabFonc pour trouver le numéro de la fonction
                        correspondante dans la table des fonctions
                construire un élément de la pile représentant un opérateur
                avancer d'une unité dans la pile
valeur est " simvar":
        appel à la fonction ConsultTabVar pour trouver la position d'une variable
                dans la table des variables
        construire un élément de la pile représentant une opérande
        avancer d'une unité dans la pile
valeur est " indvar":
        appel à la fonction ConsultTabVar pour trouver la position d'une variable
                dans la table des variables
        construire un élément de la pile représentant une opérande
        avancer d'une unité dans la pile
valeur est " const":
        construire un élément de la pile représentant une opérande
        avancer d'une unité dans la pile
valeur est " unop":
        tant que x  n'est pas nil
                appel r écursif de la procédure PileIneq
                appel à la fonction ConsultTabFonc pour trouver le numéro
                de l'opérateur unaire correspondant dans la table des fonctions
                construire un élément de la pile représentant un opérateur
                avancer d'une unité dans la pile
valeur est " binop":
        tant que x n'est pas nil
                appel récursif de procédure PileIneq
                appel à la fonction ConsultTabFonc pour trouver le numéro
                de l'opérateur binaire correspondant dans la table des fonctions
                construire un élément de la pile représentant un opérateur
                avancer d'une unité dans la pile
```

Procédure ConsultTabVar

Nom: ♦ ConsultTabVar;
Paramètres: ◊ x de type ty_constr;
Effet: • renvoie le numéro de la position d'une variable dans la table des variables;
Présentation:

test de la présence d'une variable dans la table des variables?
OUI→effectuer le numéro à la fonction *ConsultTabVar*
NON→ la valeur de variable absence est true
le message d'absence de la variable dans la table des variables est envoyé

Procédure ConsultTabFonc

Nom: ♦ ConsultTabFonc;
Paramètres: ◊ x de type ty_constr;
Effet: • renvoie le numéro de la position d'une fonction dans la table des fonctions;
Présentation:

test de la présence d'une fonction dans la table des fonctions?
OUI→ effectuer le numéro à la fonction *ConsultTabFonc*
NON→ la valeur de variable absence est true
le message d'absence de la fonction dans la table des fonctions est envoyé

4. Programmation des algorithmes

Procédure ParcourPile

Nom: ♦ ParcourPile;
Paramètres: ◊ pile de type ty_chaîne;
Effet: • construit un tableau de chaînes de dépendance statique {la variable du membre de gauche d'une inéquation correspondant à la tête d'une chaîne et toutes les variables dont elle dépend forment le reste de la chaîne};
• construit un tableau de correspondance entre le tableau des variables tab_var et la position d'une variable dans le système d'inéquations.

Procédure calcul

Nom: ♦ calcul;
Paramètres: ◊ pile de type ty_tab_in;
◊ borne de type ty_borne;
Effet: • réenvoie le résultat de calculs t;
• compteur {indique le nombre des calculs effectués };
Présentation:

```
valeur de début des calculs: borne_debut
dans la pile des inéquations
tant que le traitement de l'inéquation n'est pas terminé
        {on descend dans la pile jusqu'au dernier élément de l'inéquation}
        l'indicateur effectue le choix entre
        valeur indicateur "simvar,indvar":
                tant que pas trouvé registre libre
                        et position dans registre plus petit que longueur de registre
                        {on cherche la cellule libre de registre pour charger la valeur}
                        si cellule  registre est libre
                        alors charger valeur dans cellule
                                assigner le résultat du calcul à résultat
                        sinon avancer d'une cellule
        valeur indicateur "cons":
                tant que pas trouvé registre libre
                        et position dans registre plus petit que longueur de registre
                        {on cherche la cellule libre de registre pour charger la valeur}
                        si cellule registre est libre
                        alors charger valeur dans cellule
                                assigner le résultat du calcul à résultat
                        sinon avancer d'une cellule
        valeur indicateur "unop,binop,fonction":
                tant que pas trouvé registre libre
                        et position dans registre plus petit que longueur de registre
                        {on cherche la cellule libre de registre pour charger la valeur}
                        si cellule registre est libre
                        alors assigner les valeurs correspondantes aux opérandes
                                {on charge des valeurs des cellules appropriées}
                                effectuer calcul_opérateurs
                                {chaque opérateur a son numéro dans le tableau des fonctions,
                                en fonction de ce numéro on trouve le code des calculs à effectuer}
                                assigner le résultat du calcul à résultat
                        sinon avancer d'une cellule
                augmenter compteur d'une unité
                dépiler un élément
        assigner à t le résultat des calculs
```

Algorithme Bottom-Up

Procédure ALG1

Nom: ♦ Alg1;
Paramètres: ◊
Appel: ♥ ParcourPile {construction d'une chaîne générale, utilisée par après dans la procédure calcul};
♥ ConstrChain {construction d'une chaîne utilisée dans la partie fermeture de l'algorithme};
♥ AlgBotUp {calcul des points fixes};
Effet: • renvoie les valeurs des points fixes d'un système d'inéquations;
• renvoie le temps de calcul nécessaire pour l'algorithme Bottom- Up;
• renvoie le nombre d'opérations effectuées pour le calcul.

Procédure AlgBotUp

Nom: ♦ AlgBotUp;
Paramètres: ◊ pile de type ty_tab_in;
◊ chain_gen de type ty_chain;
◊ chain de type ty_chain;
Effet: • renvoie la table value comme résultat des calculs des points fixes d'un système d'inéquations;
Présentation:

```
{première étape du calcul}
à partir de la première inéquation jusqu'à la dernière
assigner à la borne la fourchette de calculs dans la pile des inéquations
appeler la procédure calcul
t est le résultat des calculs
si t est plus grand que la valeur de bottom
alors
        empiler la variable sur le stack
        changer la valeur de point fixe pour t

{deuxième étape des calculs}
tant que le stack n'est pas vide
        trouver la chaîne correspondant à la variable au sommet du stack
        dépiler du stack
        assigner la chaîne trouvée à c
        tant que la chaîne n'est pas vide
                assigner à borne la fourchette de calcul dans la pile des opérateurs et des opérandes
                appeler la procédure calcul
                t est le résultat de calcul
                avancer d'un élément dans chain
                si t est plus grand que valeur de point fixe précédemment trouvée
                alors
                        empiler la variable dans stack
                        changer la valeur de point fixe pour t
```

Procédure ConstrChain

Nom: ♦ ConstrChain;
Paramètres: ◊ pile de type ty_tab_in;
◊ NbPar de type integer;
Effet: • renvoie le tableau chain de type

```
chain_var=^el_chain;
    el_chain = record
    num_eq : integer;
    borne_eq : ty_borne;
    chsuiv : chain_var;
end;
ty_chain=array[1.. NbVar] of chain_var;
```

Chaque élément de la chaîne contient la suite de fourchettes qui indiquent la position des inéquations d'intérêt dans la pile. L'intérêt est de trouver des inéquations où la variable de numéro trouvé dans le tableau de chaînes figure comme opérande dans le membre de droite.

Présentation:

à partir de la première inéquation **jusqu'à** la dernière
à partir du deuxième élément de la pile **jusqu'au** dernier
si on trouve une variable qui correspond à une variable de membre gauche d'inéquation
alors assigner la valeur true à présence
si séparateur des inéquations
alors augmenter d'une unité num_eq
si présence égale true **et** inéquation est de type xi≥ fi ou xi=fi
alors construire un élément de la chaîne.

Algorithme Top-Down

Procédure Alg2

Nom: ♦ Alg2;
Paramètres: ◊
Appel: ♥ ComputeFixpoint;
Présentation:

partie interactive Homme- Machine
{on introduit la variable pour calculer le point fixe}
appeler la procédure ComputeFixpoint.

Procédure ComputeFixpoint

Nom ♦ ComputeFixpoint;
Paramètres: ◊ x de type integer;
Présentation:

initialiser des tables pt, ics, optimal, dg
appeler la procédure ConstrSuit
appeler la procédure eval.

Procédure ConstrSuit

Nom: ♦ ConstrSuit;
Paramètres: ◊ pile de type ty_tab_in;
◊ chain_gen de type ty_chain;
Effet: • renvoie la table des chaînes de dépendance statique
{la position dans le tableau correspond au numéro de la variable dans la table des variables, la suite comprend tous les éléments qui l'influencent de façon directe }
Présentation:

```
à partir du premier élément de la pile jusqu'au dernier
        si séparateur des inéquations
        alors avancer d'un élément dans la tables de chaînes
        sinon
                si valeur indicateur "simvar ou "indvar"
                alors
                        test élément est déjà compris dans la suite?
                        OUI→ assigner à variable absence valeur false
                        NON→ assigner à variable absence valeur true
                        si absenceest true
                        alors construire un nouvel élément d'une chaîne.
```

Procédure eval

Nom: ♦ eval;
Paramètres: ◊ x de type integer;
Effet: • renvoie la table pt avec les valeurs de points fixes calculées pour une variable et un groupe des variables nécessaires pour effectuer le calcul de point fixe pour le premier; les autres variables ont la valeur bottom.
Présentation:

```
si le calcul pour x n'est pas encore lancé et la valeur de variable x n'est pas optimale
alors
        assigner à ics[x] la valeur true indiquant que le calcul est initialisé
        avancer d'un élément dans la suite de dépendance statique
        si première initialisation du calcul
        alors mettre la valeur bottom dans la table pt pour la variable x
        répéter
                assigner à optimal[x] la valeur 1
                si la suite de dépendance statique n'est pas vide
                alors
                        appeler récursivement la procédure eval
                        avancer d'un élément dans la suite de dépendance statique
                        tant que la suite de dépendance statique n'est pas vide
                                appeler récursivement la procédure eval
                                appeler la procédure CreerDg
                                avancer d'un élément dans la suite de dépendance statique
                                appeler la fonction NumPar pour trouver l'inéquation
                                correspondant à la variable x
                                assigner à borne la fourchette de calcul dans la pile
                                appeler la procédure calcul
                                le résultat de calcul est résultat
```

```
                    si résultat est plus grand que pt[x]
                    {valeur de point fixe calculée auparavant }
                    alors assigner à pt[x] résultat
                    si la chaîne correspondante du graphe de dépendance n'est pas vide
                    alors appeler la procédure MettreNul
                    si la valeur de variable x n'est pas optimale
                    alors
                            assigner à ics[x] la valeur false
                            appeler récursivement la procédure eval
            jusqu'à ce que la valeur de variable x soit optimale
```

Procédure CreerDg

Nom: ♦ CreerDg;
Paramètres: ◊ y de type integer;
◊ x de type integer;
Effet: • rajoute la variable x dans la chaîne numéro y de graphe de dépendance;
de façon formelle: $dg = dg0 \setminus \{dg0(y)\} \cup \{dg0(y) \cup \{x\}\}$;
Présentation:

```
test élément x est déjà présent dans le graphe de dépendance dg[y]?
OUI→ -
NON→ ajouter élément x dans la chaîne de graphe de dépendances dg[y].
```

Procédure MettreNul

Nom: ♦ MettreNul;
Paramètres: ◊ x de type integer;
Effet: • si $dg^+(x)$ est le plus petit sous-ensemble du codomaine de dg tel que :
$\rightarrow y \in dg(x) \Rightarrow y \in dg^+(x)$
$\rightarrow y \in dg(x)$ & $z \in dg^+(y) \Rightarrow z \in dg^+(x)$
alors $dg = dg_0 \setminus \{S \in dg_0 \mid \{y_1, ..., y_n\} \cap dg_0^+(x) \neq \emptyset\}$;
Présentation:

```
assigner à pour_libere la valeur dg[x]
assigner à optimal[x] la valeur 0
mettre dg[x] à nil
initialiser la variable tete
tant que pour_libere est différent de nil
        si optimal d'une variable correspondante au premier élément
        d'une chaîne pour_libere est égale à 1
        alors
        assigner à aux1 la valeur correspondante au codomaine de dg: domaine est
        égale au premier élément d'une chaîne pour_libere
        mettre à nil un élément du graphe de dépendance: le domaine est égal
        au premier élément d'une chaîne pour_libere
        mettre optimal de même élément à 0
        assigner au suivant la valeur aux1
        tant que suivant est différent de nil
                avancer dans la chaîne suivante jusqu'au dernier élément
        si tete est différent de nil

        concaténer une chaîne correspondante à aux1 avec la chaîne tete
        avancer d'un élément dans la suite pour_libere
```

assigner à aux2 la valeur pour_libere
libérer les cellules déjà annulées
si aux2 est différent de **nil**
alors assigner à pour_libere la valeur aux2
sinon assigner à pour_libere la valeur tete
initialiser tete.

Annexe 4. BIBLIOTHEQUE DE FONCTIONS

Cette annexe présente le tableau des fonctions et des opérateurs disponibles pour utiliser dans des systèmes d'inéquations.

numéro dans table	représentation d'une fonction	le nom d'une fonction ou d'un opérateur
1	*	fois
2	div	quotient
3	mod	modulo
4	+	plus
5	-	moins
6	=	egal
7	<>	different
8	<	inferieur
9	>	superieur
10	<=	infeg
11	>=	supeg
12	and	et
13	or	ou
14	not	non
15	max	maximum
16	min	minimum
17	comp	comparaison

- la première colonne indique le numéro d'une fonction ou d'un opérateur dans la table des fonctions,
- la deuxième colonne montre sous quelle forme la fonction ou l'opérateur figure dans le programme SYSTINEQ,
- la troisième colonne définit le nom d'une fonction ou d'un opérateur.